AN INTRODUCTION TO

SEPARATION LOGIC

1. An Overview

John C. Reynolds
Carnegie Mellon University
Marktoberdorf August 15, 2008

A Program for In-place List Reversal

$$LREV \stackrel{\text{def}}{=} \mathsf{j} := \mathbf{nil};$$
$$\mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).$$

To prove $\{\mathsf{list}\,\alpha\,\mathsf{i}\}\ LREV\ \{\mathsf{list}\,\alpha^{\dagger}\,\mathsf{j}\}$, the invariant

$$\exists \alpha, \beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta,$$

(where $\mathsf{list}\,\epsilon\,\mathsf{i} \stackrel{\text{def}}{=} \mathsf{i} = \mathbf{nil}$ and $\mathsf{list}(\mathsf{a}{\cdot}\alpha)\,\mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\,\alpha\,\mathsf{j}$)
is inadequate.

An adequate invariant (in Hoare logic):

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta)$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i}, \mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j}, \mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}).$$

An adequate invariant (in separation logic):

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \ast \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta.$$

where $\ast$ is the *separating conjunction*.

To prove $\{\mathsf{list}\,\alpha\,\mathsf{i}\ *\ \mathsf{list}\,\gamma\,\mathsf{x}\}\,LREV\,\{\mathsf{list}\,\alpha^\dagger\,\mathsf{j}\ *\ \mathsf{list}\,\gamma\,\mathsf{x}\}$ in Hoare logic, we need the stronger invariant:

$$
\begin{aligned}
&(\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,\mathsf{i}\wedge\mathsf{list}\,\beta\,\mathsf{j}\wedge\alpha_0^\dagger=\alpha^\dagger{\cdot}\beta)\\
&\qquad\wedge(\forall\mathsf{k}.\ \mathbf{reachable}(\mathsf{i},\mathsf{k})\wedge\mathbf{reachable}(\mathsf{j},\mathsf{k})\Rightarrow\mathsf{k}=\mathbf{nil})\\
&\qquad\wedge\mathsf{list}\,\gamma\,\mathsf{x}\\
&\qquad\wedge(\forall\mathsf{k}.\ \mathbf{reachable}(\mathsf{x},\mathsf{k})\\
&\qquad\qquad\qquad\wedge(\mathbf{reachable}(\mathsf{i},\mathsf{k})\vee\mathbf{reachable}(\mathsf{j},\mathsf{k}))\Rightarrow\mathsf{k}=\mathbf{nil}).
\end{aligned}
$$

But in separation logic, we can use:

$$(\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,\mathsf{i}\ *\ \mathsf{list}\,\beta\,\mathsf{j}\ *\ \mathsf{list}\,\gamma\,\mathsf{x})\wedge\alpha_0^\dagger=\alpha^\dagger{\cdot}\beta).$$

Framing

Actually, in separation logic, from

$$\{\textsf{list}\,\alpha\,\textsf{i}\}\,LREV\,\{\textsf{list}\,\alpha^{\dagger}\,\textsf{j}\},$$

we can use the *frame rule* to infer directly that

$$\{\textsf{list}\,\alpha\,\textsf{i}\,*\,\textsf{list}\,\gamma\,\textsf{x}\}\,LREV\,\{\textsf{list}\,\alpha^{\dagger}\,\textsf{j}\,*\,\textsf{list}\,\gamma\,\textsf{x}\}.$$

Overview of Separation Logic

- Low-level programming language
  - Extension of simple imperative language
  - Commands for allocating, accessing, mutating, and deallocating data structures
  - Dangling pointer faults (if pointer is dereferenced)
- Program specification and proof
  - Extension of Hoare logic
  - Separating (independent, spatial) conjunction ($*$) and implication ($-\!\!*$)
- Inductive definitions over abstract structures

Early History

- Distinct Nonrepeating Tree Systems
  (Burstall 1972)
- Adding Separating Conjunction to Hoare Logic
  (Reynolds 1999, with flaws)
- Bunched Implication (BI) Logics
  (O'Hearn and Pym 1999)
- Intuitionistic Separation Logic
  (Ishtiaq and O'Hearn 2001, Reynolds 2000)
- Classical Separation Logic (Ishtiaq and O'Hearn 2001)
- Adding Address Arithmetic (Reynolds 2001)

States

Without address arithmetic (old version):

$$\mathsf{Values} = \mathsf{Integers} \cup \mathsf{Atoms} \cup \mathsf{Addresses}$$

where Integers, Atoms, and Addresses are disjoint

$$\mathbf{nil} \in \mathsf{Atoms}$$

$$\mathsf{Stores}_V = V \to \mathsf{Values}$$

$$\mathsf{Heaps} = \bigcup_{A \overset{\mathsf{fin}}{\subseteq} \mathsf{Addresses}} (A \to \mathsf{Values}^+)$$

$$\mathsf{States}_V = \mathsf{Stores}_V \times \mathsf{Heaps}$$

where V is a finite set of variables.

With address arithmetic (new version):

$$
\begin{aligned}
\mathsf{Values} &= \mathsf{Integers} \\
\mathsf{Atoms} \cup \mathsf{Addresses} &\subseteq \mathsf{Integers} \\
\text{where } &\mathsf{Atoms} \text{ and } \mathsf{Addresses} \text{ are disjoint} \\
\mathbf{nil} &\in \mathsf{Atoms} \\
\mathsf{Stores}_V &= V \to \mathsf{Values} \\
\mathsf{Heaps} &= \bigcup_{A \overset{\text{fin}}{\subseteq} \mathsf{Addresses}} (A \to \mathsf{Values}) \\
\mathsf{States}_V &= \mathsf{Stores}_V \times \mathsf{Heaps} \\
&\text{where } V \text{ is a finite set of variables.}
\end{aligned}
$$

(We assume that all but a finite number of nonnegative integers are addresses.)

The Programming Language: An Informal View

The simple imperative language:

$$:= \qquad \mathbf{skip} \qquad ; \qquad \mathbf{if} - \mathbf{then} - \mathbf{else} - \qquad \mathbf{while} - \mathbf{do} -$$

plus:

		Store : x: 3, y: 4 Heap : empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)\ ;$	⇓
		Store : x: 37, y: 4 Heap : 37: 1, 38: 2
Lookup	$\mathsf{y} := [\mathsf{x}]\ ;$	⇓
		Store : x: 37, y: 1 Heap : 37: 1, 38: 2
Mutation	$[\mathsf{x} + 1] := 3\ ;$	⇓
		Store : x: 37, y: 1 Heap : 37: 1, 38: 3
Deallocation	$\mathbf{dispose}(\mathsf{x} + 1)$	⇓
		Store : x: 37, y: 1 Heap : 37: 1

Note that:

- Expressions depend only upon the store.
  - no side effects or nontermination.
  - **cons** and [−] are parts of commands.
- Allocation is nondeterminate.

Memory Faults

		Store : x: 3, y: 4
		Heap : empty
Allocation	$x := \mathbf{cons}(1, 2)$;	$\Downarrow$
		Store : x: 37, y: 4
		Heap : 37: 1, 38: 2
Lookup	$y := [x]$;	$\Downarrow$
		Store : x: 37, y: 1
		Heap : 37: 1, 38: 2
Mutation	$[x + 2] := 3$;	$\Downarrow$
		abort

Faults can also be caused by out-of-range lookup or deallocation.

Assertions

Standard predicate calculus:

$$\wedge \qquad \vee \qquad \neg \qquad \Rightarrow \qquad \forall \qquad \exists$$

plus:

- $\mathbf{emp}$ (empty heap)
  The heap is empty.
- $e \mapsto e'$ (singleton heap)
  The heap contains one cell, at address e with contents e'.
- $p_1 * p_2$ (separating conjunction)
  The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.
- $p_1 -\!\!* \, p_2$ (separating implication)
  If the heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

Some Abbreviations

$$e \mapsto - \stackrel{\text{def}}{=} \exists x'.\, e \mapsto x' \quad \text{where } x' \text{ not free in } e$$

$$e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \mathbf{true}$$

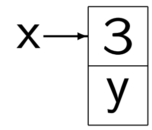
$$e \mapsto e_1, \ldots, e_n \stackrel{\text{def}}{=} e \mapsto e_1 * \cdots * e + n - 1 \mapsto e_n$$

$$e \hookrightarrow e_1, \ldots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 * \cdots * e + n - 1 \hookrightarrow e_n$$

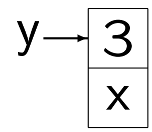
$$\text{iff } e \mapsto e_1, \ldots, e_n * \mathbf{true}$$

Examples of Separating Conjunction

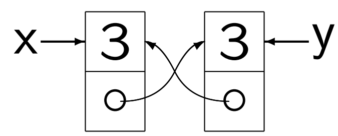
1. $x \mapsto 3, y$ asserts that x points to an adjacent pair of cells containing 3 and y.



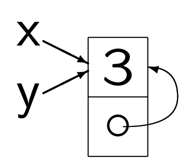
2. $y \mapsto 3, x$ asserts that y points to an adjacent pair of cells containing 3 and x.



3. $x \mapsto 3, y \ast y \mapsto 3, x$ asserts that situations (1) and (2) hold for separate parts of the heap.



4. $x \mapsto 3, y \wedge y \mapsto 3, x$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same.



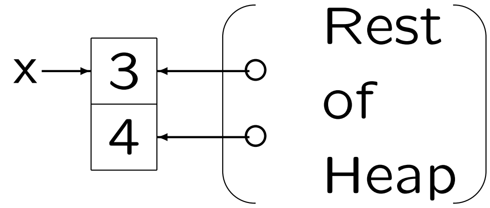
5. $x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

An Example of Separating Implication

Suppose p holds for

Store : $\mathsf{x}\colon \alpha, \ldots$

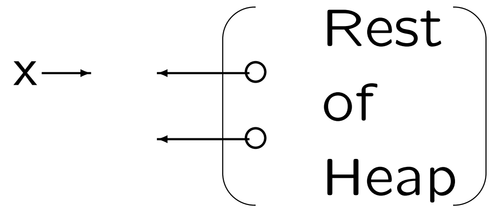
Heap : $\alpha\colon 3, \alpha + 1\colon 4, \ldots$



Then $(\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p$ holds for

Store : $\mathsf{x}\colon \alpha, \ldots$

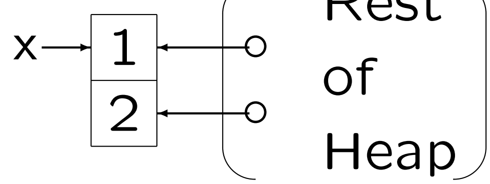
Heap : $\ldots$



and $\mathsf{x} \mapsto 1, 2 \ast ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)$ holds for

Store : $\mathsf{x}\colon \alpha, \ldots$

Heap : $\alpha\colon 1, \alpha + 1\colon 2, \ldots$



In particular,

$$\{\mathsf{x} \mapsto 1, 2 \ast ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)\}\ [\mathsf{x}] := 3\ ;\ [\mathsf{x} + 1] := 4\ \{p\},$$

and more generally,

$$\{\mathsf{x} \mapsto -, - \ast ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)\}\ [\mathsf{x}] := 3\ ;\ [\mathsf{x} + 1] := 4\ \{p\}.$$

Rules and Axiom Schemata for $*$ and $-\!\!*$

$$p_1 * p_2 \Leftrightarrow p_2 * p_1$$

$$(p_1 * p_2) * p_3 \Leftrightarrow p_1 * (p_2 * p_3)$$

$$p * \mathbf{emp} \Leftrightarrow p$$

$$(p_1 \vee p_2) * q \Leftrightarrow (p_1 * q) \vee (p_2 * q)$$

$$(p_1 \wedge p_2) * q \Rightarrow (p_1 * q) \wedge (p_2 * q)$$

$$(\exists x.\, p_1) * p_2 \Leftrightarrow \exists x.\, (p_1 * p_2) \quad \text{when } x \text{ not free in } p_2$$

$$(\forall x.\, p_1) * p_2 \Rightarrow \forall x.\, (p_1 * p_2) \quad \text{when } x \text{ not free in } p_2$$

$$\frac{p_1 \Rightarrow p_2 \qquad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2} \text{ (monotonicity)}$$

$$\frac{p_1 * p_2 \Rightarrow p_3}{p_1 \Rightarrow (p_2 -\!\!* p_3)} \text{ (currying)} \qquad \frac{p_1 \Rightarrow (p_2 -\!\!* p_3)}{p_1 * p_2 \Rightarrow p_3.} \text{ (decurrying)}$$

Two Unsound Axiom Schemata

$$p \Rightarrow p * p \quad \text{(Contraction — unsound)}$$

e.g. p : $\mathsf{x} \mapsto 1$

$$p * q \Rightarrow p \quad \text{(Weakening — unsound)}$$

e.g. p : $\mathsf{x} \mapsto 1$

q : $\mathsf{y} \mapsto 2$

Some Axiom Schemata for $\mapsto$

$$
\begin{aligned}
e_1 \mapsto e_1' \wedge e_2 \mapsto e_2' &\Leftrightarrow e_1 \mapsto e_1' \wedge e_1 = e_2 \wedge e_1' = e_2' \\
e_1 \hookrightarrow e_1' \ast e_2 \hookrightarrow e_2' &\Rightarrow e_1 \neq e_2 \\
\mathbf{emp} &\Leftrightarrow \forall x.\ \neg(x \hookrightarrow -) \\
(e \hookrightarrow e') \wedge p &\Rightarrow (e \mapsto e') \ast ((e \mapsto e') \mathrel{-\!\!\ast} p).
\end{aligned}
$$

(Regrettably, these are far from complete.)

Specifications

- $\{p\}\ c\ \{q\}$ (partial correctness)

  Starting in any state in which p holds:
  - No execution of c aborts.
  - When some execution of c terminates in a final state, then q holds in the final state.

- [p] c [q]   (total correctness)

  Starting in any state in which p holds:

  - No execution of c aborts.
  - Every execution of c terminates.
  - When some execution of c terminates in a final state, then q holds in the final state.

The Differences with Hoare Logic

- Specifications are universally quantified implicitly over both stores and heaps,
- Specifications are universally quantified implicitly over all possible executions.
- Any execution (starting in a state satisfying p) that gives a memory fault falsifies both partial and total specifications. Thus:

• • • Well-specified programs don't go wrong. • • •

— and memory-fault checking is unnecessary.

Enforcing Record Boundaries

The fact that specifications preclude memory faults acts in concert with the indeterminacy of allocation to prohibit violations of record boundaries. For example, in

$$c_0 \, ; \mathsf{x} := \mathbf{cons}(1, 2) \, ; c_1 \, ; [\mathsf{x} + 2] := 7,$$

no allocation performed by the subcommand c_0 or c_1 can be guaranteed to allocate the location $\mathsf{x} + 2$.

As long as c_0 and c_1 terminate and c_1 does not modify x, the above command may abort.

It follows that there is no postcondition that makes the specification

$$\{\mathbf{true}\} \; c_0 \, ; \mathsf{x} := \mathbf{cons}(1, 2) \, ; c_1 \, ; [\mathsf{x} + 2] := 7 \; \{?\}$$

valid.

On the Other Hand

$$\{\mathsf{x} \mapsto - \; * \; \mathsf{y} \mapsto -\}$$

$$\textbf{if } \mathsf{y} = \mathsf{x} + 1 \textbf{ then skip else}$$

$$\quad \textbf{if } \mathsf{x} = \mathsf{y} + 1 \textbf{ then } \mathsf{x} := \mathsf{y} \textbf{ else}$$

$$\qquad (\textbf{dispose}\, \mathsf{x}\, ;\, \textbf{dispose}\, \mathsf{y}\, ;\, \mathsf{x} := \textbf{cons}(1, 2))$$

$$\{\mathsf{x} \mapsto -, -\}.$$

Hoare's Inference Rules

The command-specific inference rules of Hoare logic remain sound, as do structural rules such as

- Strengthening Precedent

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\, c\, \{r\}.}$$

- Weakening Consequent

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\, c\, \{r\}.}$$

- Existential Quantification (Auxiliary Variable Elimination)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\}},$$

  where v is not free in c.

- Conjunction

$$\frac{\{p\}\ c\ \{q_1\} \qquad \{p\}\ c\ \{q_2\}}{\{p\}\ c\ \{q_1 \wedge q_2\}},$$

- Substitution

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/v_1 \to e_1,\ \ldots,\ v_n \to e_n,}$$

  where $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

The Failure of the Rule of Constancy

On the other hand,

- Rule of Constancy

$$\frac{\{p\}\ c\ \{q\}}{\{p \wedge r\}\ c\ \{q \wedge r\}},$$

 where no variable occurring free in r is modified by c.

is *unsound*, since, for example

$$\frac{\{\mathsf{x} \mapsto -\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4\}}{\{\mathsf{x} \mapsto - \wedge \mathsf{y} \mapsto 3\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4 \wedge \mathsf{y} \mapsto 3\}}$$

fails when $\mathsf{x} = \mathsf{y}$.

The Frame Rule

Instead, we have the

- Frame Rule (O'Hearn)

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}},$$

where no variable occurring free in r is modified by c.

By using the frame rule, one can extend a local specification, involving only the variables and *parts of the heap* that are actually used by c (called the *footprint* of c), by adding arbitrary predicates about variables and parts of the heap that are not touched by c.

Local Specifications

The frame rule is the key to "local reasoning" about the heap:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged. (O'Hearn)

Each valid specification $\{p\}\ c\ \{q\}$ is "tight" in the sense that it implies every cell in the footprint of c must be asserted to be active by p (or freshly allocated by c); "locality" is the converse implication that everything asserted to be active belongs to the footprint. The role of the frame rule is to infer from a local specification of a command the more global specification appropriate to the possibly larger footprint of an enclosing command.

Inference Rules for Mutation

- Local

$$\overline{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

- Global

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}.}$$

- Backward Reasoning

$$\overline{\{(e \mapsto -) \ast ((e \mapsto e') \mathrel{-\!\!\ast} p)\}\ [e] := e'\ \{p\}.}$$

Inference Rules for Deallocation

- Local

$$\overline{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}}.$$

- Global, Backwards Reasoning

$$\overline{\{(e \mapsto -) \ * \ r\}\ \mathbf{dispose}\ e\ \{r\}}.$$

Inference Rules for Nonoverwriting Allocation

- Local

$$\overline{\qquad\qquad} \atop \{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}\},$$

where v is not free in $\overline{e} \stackrel{\text{def}}{=} e_1, \ldots, e_n$.

- Global

$$\overline{\qquad\qquad} \atop \{r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e}) \ * \ r\},$$

where v is not free in $\overline{e}$ or r.

(We postpone more complex rules with quantifiers.)

An Example of an Annotated Specification: Gluing Records

$$
\begin{array}{l}
\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\
\textbf{if } \mathsf{y} = \mathsf{x} + 1 \textbf{ then} \\
\qquad \{\mathsf{x} \mapsto -, -\} \\
\qquad \textbf{skip} \\
\textbf{else if } \mathsf{x} = \mathsf{y} + 1 \textbf{ then} \\
\qquad \{\mathsf{y} \mapsto -, -\} \\
\qquad \mathsf{x} := \mathsf{y} \\
\textbf{else} \\
\qquad \big(\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\} \\
\qquad \textbf{dispose } \mathsf{x}\,; \\
\qquad \{\mathsf{y} \mapsto -\} \\
\qquad \textbf{dispose } \mathsf{y}\,; \\
\qquad \{\textbf{emp}\} \\
\qquad \mathsf{x} := \textbf{cons}(1, 2)\big) \\
\{\mathsf{x} \mapsto -, -\}.
\end{array}
$$

Another Example: Relative Pointers

$$\{\mathbf{emp}\}$$

$$\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a}) \;;$$

$$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\}$$

$$\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b}) \;;$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \;*\; (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, -) \;*\; (\mathsf{y} \mapsto \mathsf{b}, -)\}$$

$$[\mathsf{x} + 1] := \mathsf{y} - \mathsf{x} \;;$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \;*\; (\mathsf{y} \mapsto \mathsf{b}, -)\}$$

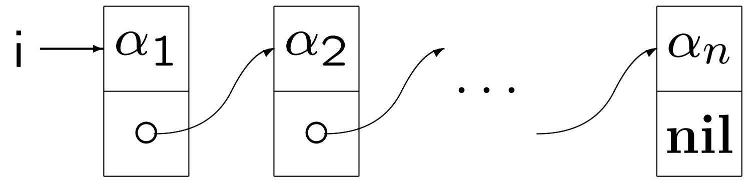
$$[\mathsf{y} + 1] := \mathsf{x} - \mathsf{y} \;;$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \;*\; (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\}$$

$$\{\exists \mathsf{o}.\; (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \;*\; (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b}, \; - \mathsf{o})\}.$$

Singly-linked Lists

list α i:



is defined by

$$\mathsf{list}\ \epsilon\ \mathsf{i} \overset{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil}$$

$$\mathsf{list}\ (\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \overset{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ * \ \mathsf{list}\ \alpha\ \mathsf{j},$$

where

- ϵ is the empty sequence.
- $\alpha{\cdot}\beta$ is the composition of α followed by β.
- $\alpha^\dagger$ is the reflection of α.

One can also derive an emptyness test:

$$\mathsf{list}\ \alpha\ \mathsf{i} \Rightarrow (\mathsf{i} = \mathbf{nil} \Leftrightarrow \alpha = \epsilon).$$

S-expressions (à la LISP)

$\tau \in$ S-exps iff

$$\begin{aligned} \tau &\in \text{Atoms} \\ \text{or } \tau &= (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exps.} \end{aligned}$$

Representing S-expressions by Trees (no sharing)

For $\tau \in$ S-exps, we define the assertion

$$\mathsf{tree}\, \tau\, (i)$$

by structural induction:

$$\mathsf{tree}\, a\, (i) \text{ iff } \mathbf{emp} \wedge i = a$$

$$\begin{aligned}&\mathsf{tree}\, (\tau_1 \cdot \tau_2)\, (i) \text{ iff}\\ &\qquad \exists i_1, i_2.\ i \mapsto i_1, i_2 \ * \ \mathsf{tree}\, \tau_1\, (i_1) \ * \ \mathsf{tree}\, \tau_2\, (i_2).\end{aligned}$$

Representing S-expressions by Dags (with sharing)

For $\tau \in$ S-exps, we define

$$\mathsf{dag}\,\tau\,(i)$$

by:

$$\mathsf{dag}\,a\,(i) \text{ iff } i = a$$

$$\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff}$$
$$\exists i_1, i_2.\ i \mapsto i_1, i_2\ *\ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)).$$

Proving the Schorr-Waite Marking Algorithm (Yang)

- We abandon address arithmetic, and require all records to contain two address fields and two boolean fields.
- Only reachable cells are in heap.

Let

$$\mathsf{allocated}(\mathsf{x}) \stackrel{\mathsf{def}}{=} \mathsf{x} \hookrightarrow -,-,-,-$$

$$\mathsf{markedR} \stackrel{\mathsf{def}}{=} \forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \mathsf{x} \hookrightarrow -,-,-,\mathbf{true}$$

$$\mathsf{noDangling}(\mathsf{x}) \stackrel{\mathsf{def}}{=} (\mathsf{x} = \mathbf{nil}) \vee \mathsf{allocated}(\mathsf{x})$$

$$\mathsf{noDanglingR} \stackrel{\mathsf{def}}{=} \forall \mathsf{x},\mathsf{l},\mathsf{r}.\ (\mathsf{x} \hookrightarrow \mathsf{l},\mathsf{r},-,-) \Rightarrow \mathsf{noDangling}(\mathsf{l}) \wedge \mathsf{noDangling}(\mathsf{r}).$$

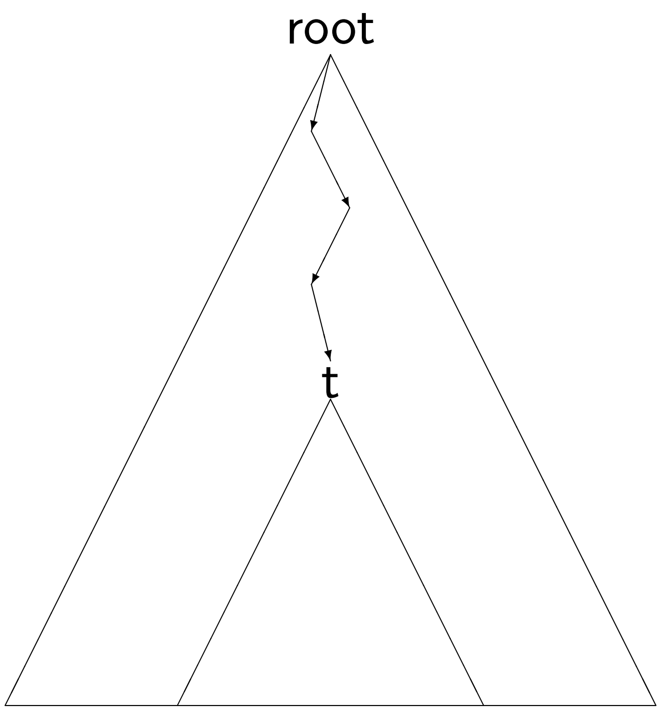
Then the invariant of the program is

$$\begin{aligned}&\mathsf{noDanglingR} \wedge \mathsf{noDangling}(\mathsf{t}) \wedge \mathsf{noDangling}(\mathsf{p}) \wedge \\ &\big(\mathsf{listMarkedNodesR}(\mathsf{stack},\mathsf{p}) \ * \\ &\quad (\mathsf{restoredlistR}(\mathsf{stack},\mathsf{t}) \mathrel{-\!\!*} \mathsf{spansR}(\mathsf{STree},\mathsf{root}))\big) \wedge \\ &\big(\mathsf{markedR} \ * \ \big(\mathsf{unmarkedR} \wedge \big(\forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \\ &\quad (\mathsf{reach}(\mathsf{t},\mathsf{x}) \vee \mathsf{reachRightChildInList}(\mathsf{stack},\mathsf{x}))\big)\big)\big).\end{aligned}$$

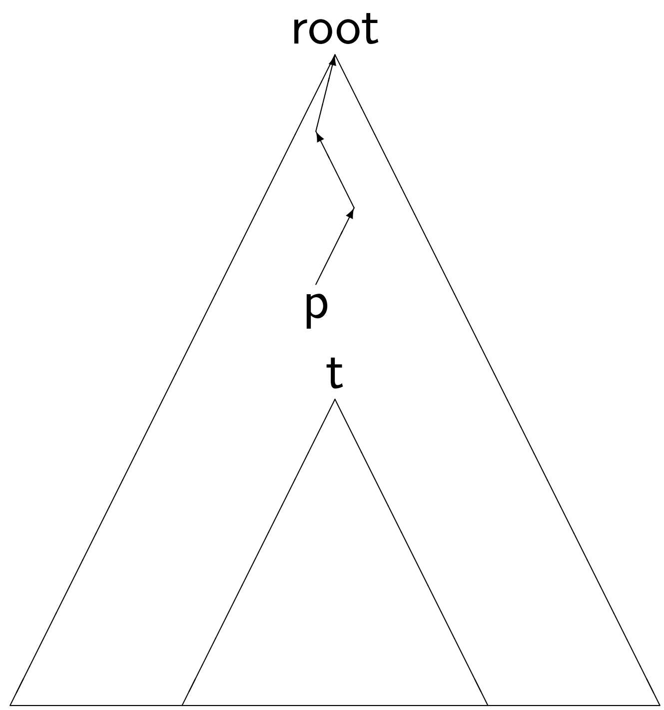
Proving Schorr-Waite (continued)

$$
\begin{array}{l}
\mathsf{noDanglingR} \wedge \mathsf{noDangling}(\mathsf{t}) \wedge \mathsf{noDangling}(\mathsf{p}) \wedge \\
\Big(\mathsf{listMarkedNodesR}(\mathsf{stack}, \mathsf{p}) \; * \\
\quad (\mathsf{restoredListR}(\mathsf{stack}, \mathsf{t}) \mathrel{-\!\!*} \mathsf{spansR}(\mathsf{STree}, \mathsf{root}))\Big) \wedge \\
\Big(\mathsf{markedR} \; * \; \Big(\mathsf{unmarkedR} \wedge \Big(\forall \mathsf{x}.\ \mathsf{allocated}(\mathsf{x}) \Rightarrow \\
\quad (\mathsf{reach}(\mathsf{t}, \mathsf{x}) \vee \mathsf{reachRightChildInList}(\mathsf{stack}, \mathsf{x}))\Big)\Big)\Big).
\end{array}
$$

restoredListR(stack, t):



listMarkedNodesR(stack, p):



Shared-Variable Concurrency (O'Hearn and Brookes)

Without Critical Regions

Hoare (1972):

$$\frac{\{p_1\}\ c_1\ \{q_1\} \qquad \{p_2\}\ c_2\ \{q_2\}}{\{p_1 \wedge p_2\}\ c_1 \parallel c_2\ \{q_1 \wedge q_2\},}$$

when the free variables of p_1, c_1, and q_1 are not modified by c_2, and vice-versa.

O'Hearn (2002):

$$\frac{\{p_1\}\ c_1\ \{q_1\} \qquad \{p_2\}\ c_2\ \{q_2\}}{\{p_1 \ast p_2\}\ c_1 \parallel c_2\ \{q_1 \ast q_2\}}$$

(with the same side condition as above).

With Critical Regions: A Simple Buffer

$$\{\mathbf{emp}\}$$

$$\{\mathbf{emp} * \mathbf{emp}\}$$

$$\begin{array}{lcl}
\{\mathbf{emp}\} & & \{\mathbf{emp}\} \\
\mathsf{x} := \mathbf{cons}(\ldots,\ldots)\,; & & \mathsf{get(y)}\,; \\
\{\mathsf{x} \mapsto -,-\} & \| & \{\mathsf{y} \mapsto -,-\} \\
\mathsf{put(x)}\,; & & \text{"Use y"}\,; \\
\{\mathbf{emp}\} & & \{\mathsf{y} \mapsto -,-\} \\
 & & \mathbf{dispose}\,\mathsf{y}\,; \\
 & & \{\mathbf{emp}\}
\end{array}$$

$$\{\mathbf{emp} * \mathbf{emp}\}$$

$$\{\mathbf{emp}\}$$

Behind the scenes:

$$\mathsf{put(x)} = \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \neg\,\mathsf{full}\ \mathbf{do}\,(\mathsf{c} := \mathsf{x}\,; \mathsf{full} := \mathbf{true})$$

$$\mathsf{get(y)} = \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \mathsf{full}\ \mathbf{do}\,(\mathsf{y} := \mathsf{c}\,; \mathsf{full} := \mathbf{false})$$

The Resource Invariant

$$\mathsf{R} \stackrel{\text{def}}{=} (\mathsf{full} \wedge \mathsf{c} \mapsto -,-) \vee (\neg\,\mathsf{full} \wedge \mathbf{emp}).$$

$\mathsf{put}(\mathsf{x}) =$
$\quad \{\mathsf{x} \mapsto -,-\}$
$\quad \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \neg\,\mathsf{full}\ \mathbf{do}\ \Big($
$\qquad \{(R \ast \mathsf{x} \mapsto -,-) \wedge \neg\,\mathsf{full}\}$
$\qquad \{\mathbf{emp} \ast \mathsf{x} \mapsto -,-\}$
$\qquad \{\mathsf{x} \mapsto -,-\}$
$\qquad \mathsf{c} := \mathsf{x}\,;\ \mathsf{full} := \mathbf{true}$
$\qquad \{\mathsf{full} \wedge \mathsf{c} \mapsto -,-\}$
$\qquad \{R\}$
$\qquad \{R \ast \mathbf{emp}\}\Big)$
$\quad \{\mathbf{emp}\}$

$\mathsf{get}(\mathsf{y}) =$
$\quad \{\mathbf{emp}\}$
$\quad \mathbf{with}\ \mathsf{buf}\ \mathbf{when}\ \mathsf{full}\ \mathbf{do}\ \Big($
$\qquad \{(R \ast \mathbf{emp}) \wedge \mathsf{full}\}$
$\qquad \{\mathsf{c} \mapsto -,- \ast \mathbf{emp}\}$
$\qquad \{\mathsf{c} \mapsto -,-\}$
$\qquad \mathsf{y} := \mathsf{c}\,;\ \mathsf{full} := \mathbf{false}$
$\qquad \{\neg\,\mathsf{full} \wedge \mathsf{y} \mapsto -,-\}$
$\qquad \{(\neg\,\mathsf{full} \wedge \mathbf{emp}) \ast \mathsf{y} \mapsto -,-\}$
$\qquad \{R \ast \mathsf{y} \mapsto -,-\}\Big)$
$\quad \{\mathsf{y} \mapsto -,-\}$

The Overall Program

$\{R * \mathbf{emp}\}$

resource buf **in**

$$\begin{array}{ccc} & \{\mathbf{emp}\} & \\ & \{\mathbf{emp} * \mathbf{emp}\} & \\ \vdots & \| & \vdots \\ & \{\mathbf{emp} * \mathbf{emp}\} & \\ & \{\mathbf{emp}\} & \end{array}$$

$\{R * \mathbf{emp}\}$

Fractional Permissions (Bornat, following Boyland)

We write $e \mapsto_z e'$, where z is a real number such that $0 < z \leq 1$, to assert e points to e' with permission z.

- $e \mapsto_1 e'$ is the same as $e \mapsto e'$, so that a permission of one allows all operations.
- Only lookup is allowed when $z < 1$.

Then

$$e \mapsto_z e' \ast e \mapsto_{z'} e' \text{ iff } e \mapsto_{z+z'} e'$$

and

$$\begin{gathered}
\{\mathbf{emp}\} v := \mathbf{cons}(e_1, \ldots, e_n) \{e \mapsto_1 e_1, \ldots, e_n\} \\
\{e \mapsto_1 -\} \mathbf{dispose}(e) \{\mathbf{emp}\} \\
\{e \mapsto_1 -\} [e] := e' \{e \mapsto_1 e'\} \\
\{e \mapsto_z e'\} v := [e] \{e \mapsto_z e' \wedge v = e\},
\end{gathered}$$

with appropriate restrictions on variable occurrences.

AN INTRODUCTION TO

SEPARATION LOGIC

2. Assertions

John C. Reynolds
Carnegie Mellon University
Marktoberdorf August 15, 2008

Some Notation for Functions

We write

$$[\, x_1 \colon y_1 \mid \ldots \mid x_n \colon y_n \,]$$

for the function with domain $\{x_1, \ldots, x_n\}$ that maps each x_i into y_i, and

$$[\, f \mid x_1 \colon y_1 \mid \ldots \mid x_n \colon y_n \,]$$

for the function whose domain is the union of the domain of f with $\{x_1, \ldots, x_n\}$, that maps each x_i into y_i and all other members x of the domain of f into $f\,x$.

For heaps, we write

$$h_0 \perp h_1$$

when h_0 and h_1 have disjoint domains, and

$$h_0 \cdot h_1$$

to denote the union of heaps with disjoint domains.

Free Variables

For any phrase p,

FV(p) denotes the set of variables occurring free in p.

There are no binding constructions in expressions or boolean expressions, so that for these phrases FV(e) is the set of all variables occurring in e. In assertions, quantifiers are binding constructions. In commands, declarations will be binding constructions.

The scope of a binding construction is the phrase immediately following the binding occurrence of a variable, except in

$$\mathbf{newvar}\ v = \underline{e}\ \mathbf{in}\ c,$$

where the underline phrases are excluded from the scope.

Total Substitution

For any phrase p such that FV(p) $\subseteq \{v_1, \ldots, v_n\}$, we write

$$p/v_1 \to e_1, \ldots, v_n \to e_n$$

to denote the phrase obtained from p by simultaneously substituting each expression e_i for the variable v_i, (When there are bound variables in p, they will be renamed to avoid capture.)

The Total Substitution Law for Expressions

Proposition 1 *Let* δ *abbreviate the substitution*

$$v_1 \to e_1, \ldots, v_n \to e_n,$$

let s *be a store such that* $\mathsf{FV}(e_1) \cup \cdots \cup \mathsf{FV}(e_n) \subseteq \mathsf{dom}\, s$*, and let*

$$\widehat{s} = [\, v_1 \colon [\![e_1]\!]_{\mathsf{exp}} s \mid \ldots \mid v_n \colon [\![e_n]\!]_{\mathsf{exp}} s \,].$$

If e *is an expression (or boolean expression) such that* $\mathsf{FV}(e) \subseteq \{v_1, \ldots, v_n\}$*, then*

$$[\![e/\delta]\!]_{\mathsf{exp}} s = [\![e]\!]_{\mathsf{exp}} \widehat{s}.$$

Partial Substitution

When FV(p) is not a subset of $\{v_1, \ldots, v_n\}$,

$$p/v_1 \to e_1, \ldots, v_n \to e_n$$

abbreviates

$$p/v_1 \to e_1, \ldots, v_n \to e_n, v'_1 \to v'_1, \ldots, v'_k \to v'_k,$$

where $\{v'_1, \ldots, v'_k\} = \mathsf{FV}(p) - \{v_1, \ldots, v_n\}$.

The Meaning of Assertions

When s is a store, h is a heap, and p is an assertion whose free variables belong to the domain of s, we write

$$s, h \vDash p$$

to indicate that the state s, h *satisfies* p, or p *is true in* s, h, or p *holds in* s, h. Then:

$$\begin{aligned}
s, h \vDash b \ &\text{iff}\ \ [\![b]\!]_{\mathsf{boolexp}} s = \mathbf{true}, \\
s, h \vDash \neg\, p \ &\text{iff}\ \ s, h \vDash p \text{ is false}, \\
s, h \vDash p_0 \wedge p_1 \ &\text{iff}\ \ s, h \vDash p_0 \text{ and } s, h \vDash p_1 \\
&\qquad\qquad \text{(and similarly for } \vee,\ \Rightarrow,\ \Leftrightarrow\text{)}, \\
s, h \vDash \forall v.\ p \ &\text{iff}\ \ \forall x \in \mathbf{Z}.\ [\, s \mid v\colon x\,], h \vDash p, \\
s, h \vDash \exists v.\ p \ &\text{iff}\ \ \exists x \in \mathbf{Z}.\ [\, s \mid v\colon x\,], h \vDash p, \\
s, h \vDash \mathbf{emp} \ &\text{iff}\ \ \mathsf{dom}\, h = \{\}, \\
s, h \vDash e \mapsto e' \ &\text{iff}\ \ \mathsf{dom}\, h = \{[\![e]\!]_{\mathsf{exp}} s\} \text{ and} \\
&\qquad\qquad h([\![e]\!]_{\mathsf{exp}} s) = [\![e']\!]_{\mathsf{exp}} s, \\
s, h \vDash p_0 \ast p_1 \ &\text{iff}\ \ \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and} \\
&\qquad\qquad s, h_0 \vDash p_0 \text{ and } s, h_1 \vDash p_1, \\
s, h \vDash p_0 \mathrel{-\!\!*} p_1 \ &\text{iff}\ \ \forall h'.\ (h' \perp h \text{ and } s, h' \vDash p_0) \text{ implies} \\
&\qquad\qquad s, h \cdot h' \vDash p_1.
\end{aligned}$$

When $s, h \vDash p$ holds for all states s, h (such that the domain of s contains the free variables of p), we say that p is *valid*.

When $s, h \vDash p$ holds for some state s, h, we say that p is *satisfiable*.

For Instance

$$
\begin{aligned}
s, h \vDash \mathsf{x} \mapsto 0 \ * \ \mathsf{y} \mapsto 1 \\
\text{iff } & \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \\
& \text{and } s, h_0 \vDash \mathsf{x} \mapsto 0 \\
& \text{and } s, h_1 \vDash \mathsf{y} \mapsto 1 \\
\text{iff } & \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \\
& \text{and } \operatorname{dom} h_0 = \{s\,\mathsf{x}\} \text{ and } h_0(s\,\mathsf{x}) = 0 \\
& \text{and } \operatorname{dom} h_1 = \{s\,\mathsf{y}\} \text{ and } h_1(s\,\mathsf{y}) = 1 \\
\text{iff } & s\,\mathsf{x} \neq s\,\mathsf{y} \\
& \text{and } \operatorname{dom} h = \{s\,\mathsf{x}, s\,\mathsf{y}\} \\
& \text{and } h(s\,\mathsf{x}) = 0 \text{ and } h(s\,\mathsf{y}) = 1 \\
\text{iff } & s\,\mathsf{x} \neq s\,\mathsf{y} \text{ and } h = [\, s\,\mathsf{x}\colon 0 \mid s\,\mathsf{y}\colon 1\,].
\end{aligned}
$$

Examples

$$
\begin{aligned}
s, h \vDash \mathsf{x} \mapsto \mathsf{y} &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}\} \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y} \\
s, h \vDash \mathsf{x} \mapsto - &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}\} \\
s, h \vDash \mathsf{x} \hookrightarrow \mathsf{y} &\text{ iff } s\,\mathsf{x} \in \operatorname{dom} h \text{ and } h(s\,\mathsf{x}) = s\,\mathsf{y} \\
s, h \vDash \mathsf{x} \hookrightarrow - &\text{ iff } s\,\mathsf{x} \in \operatorname{dom} h \\
s, h \vDash \mathsf{x} \mapsto \mathsf{y}, \mathsf{z} &\text{ iff } h = [\, s\,\mathsf{x}\colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1\colon s\,\mathsf{z} \,] \\
s, h \vDash \mathsf{x} \mapsto -, - &\text{ iff } \operatorname{dom} h = \{s\,\mathsf{x}, s\,\mathsf{x} + 1\} \\
s, h \vDash \mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{z} &\text{ iff } h \supseteq [\, s\,\mathsf{x}\colon s\,\mathsf{y} \mid s\,\mathsf{x} + 1\colon s\,\mathsf{z} \,] \\
s, h \vDash \mathsf{x} \hookrightarrow -, - &\text{ iff } \operatorname{dom} h \supseteq \{s\,\mathsf{x}, s\,\mathsf{x} + 1\}.
\end{aligned}
$$

More Examples of $*$

Suppose $s\,\mathsf{x}$ and $s\,\mathsf{y}$ are distinct addresses, so that

$$h_0 = [\, s\,\mathsf{x}\colon 0\,] \qquad \text{and} \qquad h_1 = [\, s\,\mathsf{y}\colon 1\,]$$

are heaps with disjoint domains. Then

If p is:	then $s, h \vDash p$ iff:
$\mathsf{x} \mapsto 0$	$h = h_0$
$\mathsf{y} \mapsto 1$	$h = h_1$
$\mathsf{x} \mapsto 0 \, * \, \mathsf{y} \mapsto 1$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \, * \, \mathsf{x} \mapsto 0$	$\mathbf{false}$
$\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1$	$h = h_0$ or $h = h_1$
$\mathsf{x} \mapsto 0 \, * \, (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$h = h_0 \cdot h_1$
$(\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1) \, * \, (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$h = h_0 \cdot h_1$
$\mathsf{x} \mapsto 0 \, * \, \mathsf{y} \mapsto 1 \, * \, (\mathsf{x} \mapsto 0 \vee \mathsf{y} \mapsto 1)$	$\mathbf{false}$
$\mathsf{x} \mapsto 0 \, * \, \mathbf{true}$	$h_0 \subseteq h$
$\mathsf{x} \mapsto 0 \, * \, \neg\, \mathsf{x} \mapsto 0$	$h_0 \subseteq h.$

Inference Rules

$$\frac{\mathcal{P}_1 \qquad \cdots \qquad \mathcal{P}_n}{\mathcal{C}}$$

(zero or more premisses)
(one conclusion)

Inference

Inference Rules	Instances
$\dfrac{p_0 \qquad p_0 \Rightarrow p_1}{p_1}$	$\dfrac{\mathsf{x} + 0 = \mathsf{x} \qquad \mathsf{x} + 0 = \mathsf{x} \Rightarrow \mathsf{x} = \mathsf{x} + 0}{\mathsf{x} = \mathsf{x} + 0}$
$e_2 = e_1 \Rightarrow e_1 = e_2$	$\mathsf{x} + 0 = \mathsf{x} \Rightarrow \mathsf{x} = \mathsf{x} + 0$
$\mathsf{x} + 0 = \mathsf{x}$	$\mathsf{x} + 0 = \mathsf{x}$

A Proof

$$\mathsf{x} + 0 = \mathsf{x}$$
$$\mathsf{x} + 0 = \mathsf{x} \Rightarrow \mathsf{x} = \mathsf{x} + 0$$
$$\mathsf{x} = \mathsf{x} + 0.$$

Notice:

- Metavariables are in italics (or Greek), object variables are in sans serif.
- An inference rule is *sound* iff, for every instance, if the premisses are all valid, then the conclusion is valid.
- An *axiom schema* is an inference rule with zero premisses.
- An *axiom* is an axiom schema with no metavariables.

A Subtlety

$$\frac{p}{q}$$ is sound iff, whenever p is valid, q is valid.

$$\frac{}{p \Rightarrow q}$$ is sound iff $p \Rightarrow q$ is valid.

For example,

$$\frac{p}{\forall v.\ p} \quad \text{e.g.} \quad \frac{\mathsf{x} = 0}{\forall \mathsf{x}.\ \mathsf{x} = 0}$$

is sound, but

$$p \Rightarrow \forall v.\ p \quad \text{e.g.} \quad \mathsf{x} = 0 \Rightarrow \forall \mathsf{x}.\ \mathsf{x} = 0$$

is not valid.

Inference Rules for Predicate Logic

$$\frac{p \qquad p \Rightarrow q}{q} \quad \text{(modus ponens)}$$

$$\frac{p \Rightarrow q}{p \Rightarrow (\forall v.\ q)} \quad \text{when } v \notin \mathsf{FV}(p)$$

$$\frac{p \Rightarrow q}{(\exists v.\ p) \Rightarrow q} \quad \text{when } v \notin \mathsf{FV}(q).$$

Axiom Schema

$$
\begin{aligned}
p &\Rightarrow (q \Rightarrow p) \\
(p \Rightarrow (q \Rightarrow r)) &\Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)) \\
(p \wedge q) &\Rightarrow p \\
(p \wedge q) &\Rightarrow q \\
p &\Rightarrow (q \Rightarrow (p \wedge q)) \\
p &\Rightarrow (p \vee q) \\
q &\Rightarrow (p \vee q) \\
(p \Rightarrow r) &\Rightarrow ((q \Rightarrow r) \Rightarrow ((p \vee q) \Rightarrow r)) \\
(p \Rightarrow q) &\Rightarrow ((p \Rightarrow \neg\, q) \Rightarrow \neg\, p) \\
\neg(\neg\, p) &\Rightarrow p \\
(p \Leftrightarrow q) &\Rightarrow ((p \Rightarrow q) \wedge (q \Rightarrow p)) \\
((p \Rightarrow q) \wedge (q \Rightarrow p)) &\Rightarrow (p \Leftrightarrow q) \\
(\forall v.\ p) &\Rightarrow (p/v \to e) \\
(p/v \to e) &\Rightarrow (\exists v.\ p).
\end{aligned}
$$

Inference Rules for $*$ and $-\!\!*$

$$p_0 * p_1 \Leftrightarrow p_1 * p_0$$

$$(p_0 * p_1) * p_2 \Leftrightarrow p_0 * (p_1 * p_2)$$

$$p * \mathbf{emp} \Leftrightarrow p$$

$$(p_0 \vee p_1) * q \Leftrightarrow (p_0 * q) \vee (p_1 * q)$$

$$(p_0 \wedge p_1) * q \Rightarrow (p_0 * q) \wedge (p_1 * q)$$

$$(\exists x.\, p_0) * p_1 \Leftrightarrow \exists x.\, (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1$$

$$(\forall x.\, p_0) * p_1 \Rightarrow \forall x.\, (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1$$

$$\frac{p_0 \Rightarrow p_1 \qquad q_0 \Rightarrow q_1}{p_0 * q_0 \Rightarrow p_1 * q_1} \quad \text{(monotonicity)}$$

$$\frac{p_0 * p_1 \Rightarrow p_2}{p_0 \Rightarrow (p_1 -\!\!* p_2)} \text{ (currying)} \qquad \frac{p_0 \Rightarrow (p_1 -\!\!* p_2)}{p_0 * p_1 \Rightarrow p_2.} \text{ (decurrying)}$$

Some Axiom Schemata for $\mapsto$ and $\hookrightarrow$

$$e_0 \mapsto e_0' \wedge e_1 \mapsto e_1' \Leftrightarrow e_0 \mapsto e_0' \wedge e_0 = e_1 \wedge e_0' = e_1'$$

$$e_0 \hookrightarrow e_0' * e_1 \hookrightarrow e_1' \Rightarrow e_0 \neq e_1$$

$$\mathbf{emp} \Leftrightarrow \forall x.\ \neg(x \hookrightarrow -)$$

$$(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') * ((e \mapsto e') -\!\!* \, p).$$

Pure Assertions

An assertion p is *pure* iff, for all stores s and all heaps h and h',

$$s, h \vDash p \text{ iff } s, h' \vDash p.$$

A sufficient syntactic criteria is that an assertion is pure if it does not contain $\mathbf{emp}$, $\mapsto$, or $\hookrightarrow$.

Axiom Schemata for Purity

$p_0 \wedge p_1 \Rightarrow p_0 * p_1$	when p_0 or p_1 is pure
$p_0 * p_1 \Rightarrow p_0 \wedge p_1$	when p_0 and p_1 are pure
$(p \wedge q) * r \Leftrightarrow (p * r) \wedge q$	when q is pure
$(p_0 \mathrel{-\!\!*} p_1) \Rightarrow (p_0 \Rightarrow p_1)$	when p_0 is pure
$(p_0 \Rightarrow p_1) \Rightarrow (p_0 \mathrel{-\!\!*} p_1)$	when p_0 and p_1 are pure.

Precise Assertions

An assertion q is *precise* iff

For all s and h, there is at most one $h' \subseteq h$ such that

$$s, h' \vDash q.$$

Examples of Precise Assertions

- $e \mapsto -$.
- $p * q$, when p and q are precise.
- $p \wedge q$, when p or q is precise.
- p, when $p \Rightarrow q$ is valid and q is precise.
- $\mathsf{list}\,\alpha\,e$ and $\exists\alpha.\ \mathsf{list}\,\alpha\,e$.
- $\mathsf{tree}\,\tau\,(e)$ and $\exists\tau.\ \mathsf{tree}\,\tau\,(e)$.

Examples of Imprecise Assertions

- $\mathbf{true}$
- $\mathbf{emp} \vee \mathsf{x} \mapsto 10$
- $\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 10$
- $\exists \mathsf{x}.\ \mathsf{x} \mapsto 10$
- $\mathsf{dag}\ \tau\ (\mathsf{i})$
- $\exists \tau.\ \mathsf{dag}\ \tau\ (\mathsf{i})$

Preciseness and Distributivity

The semi-distributive laws

$$(p_0 \wedge p_1) \ast q \Rightarrow (p_0 \ast q) \wedge (p_1 \ast q)$$
$$(\forall x.\, p) \ast q \Rightarrow \forall x.\, (p \ast q) \quad \text{when } x \text{ not free in } q$$

are valid for all assertions. But their converses

$$(p_0 \ast q) \wedge (p_1 \ast q) \Rightarrow (p_0 \wedge p_1) \ast q$$
$$\forall x.\, (p \ast q) \Rightarrow (\forall x.\, p) \ast q \quad \text{when } x \text{ not free in } q$$

are not. For example, when

$$s(\mathsf{x}) = 1 \qquad s(\mathsf{y}) = 2 \qquad h = [\, 1\colon 10 \mid 2\colon 20 \,],$$

the assertion

$$(\mathsf{x} \mapsto 10 \ast (\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 20)) \wedge (\mathsf{y} \mapsto 20 \ast (\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 20))$$

is true, but

$$((\mathsf{x} \mapsto 10 \wedge \mathsf{y} \mapsto 20) \ast (\mathsf{x} \mapsto 10 \vee \mathsf{y} \mapsto 20))$$

is false.

However, the converses are valid when q is precise.

Preciseness and Distributivity (continued)

Proposition 2 *When q is precise,*

$$(p_0 \ * \ q) \wedge (p_1 \ * \ q) \Rightarrow (p_0 \wedge p_1) \ * \ q$$

is valid. When q is precise and x is not free in q,

$$\forall x. \, (p \ * \ q) \Rightarrow (\forall x. \, p) \ * \ q$$

is valid.

Proof (first law) Suppose $s, h \vDash (p_0 \ * \ q) \wedge (p_1 \ * \ q)$. Then there are:

- An $h_0 \subseteq h$ such that $s, h - h_0 \vDash p_0$ and $s, h_0 \vDash q$, and
- An $h_1 \subseteq h$ such that $s, h - h_1 \vDash p_1$ and $s, h_1 \vDash q$.

Thus, since q is precise,

$$h_0 = h_1$$
$$h - h_0 = h - h_1$$
$$s, h - h_0 \vDash p_0 \wedge p_1$$
$$s, h \vDash (p_0 \wedge p_1) \ * \ q.$$

end of proof

Intuitionistic Assertions

An assertion i is *intuitionistic* iff, for all stores s and heaps h and h':

$$(h \subseteq h' \text{ and } s, h \vDash i) \text{ implies } s, h' \vDash i.$$

Assume i and i' are intuitionistic assertions, and p is any assertion. Then:

- The following assertions are intuitionistic:

Any pure assertion	$p * i$
$p \mathrel{-\!\!*} i$	$i \mathrel{-\!\!*} p$
$i \wedge i'$	$i \vee i'$
$\forall v.\ i$	$\exists v.\ i$
$\mathsf{dag}\ \tau\ (e)$	$\exists \tau.\ \mathsf{dag}\ \tau\ (e)$,

  and as special cases:

  $$p * \mathbf{true} \qquad \mathbf{true} \mathrel{-\!\!*} p \qquad e \hookrightarrow e'.$$

- The following inference rules are sound:

$$(i \ast i') \Rightarrow (i \wedge i')$$

$$(i \ast p) \Rightarrow i \qquad\qquad i \Rightarrow (p \mathrel{-\!\!\ast} i)$$

$$\frac{p \Rightarrow i}{(p \ast \mathbf{true}) \Rightarrow i} \qquad \frac{i \Rightarrow p}{i \Rightarrow (\mathbf{true} \mathrel{-\!\!\ast} p).}$$

The last two of these rules, in conjunction with the rules

$$p \Rightarrow (p \ast \mathbf{true}) \qquad\qquad (\mathbf{true} \mathrel{-\!\!\ast} p) \Rightarrow p,$$

which hold for all assertions, imply that

- $p \ast \mathbf{true}$ is the strongest intuitionistic assertion weaker than p.
- $\mathbf{true} \mathrel{-\!\!\ast} p$ is the weakest intuitionistic assertion that is stronger than p.
- $i \Leftrightarrow (i \ast \mathbf{true})$.
- $(\mathbf{true} \mathrel{-\!\!\ast} i) \Leftrightarrow i$.

The Intuitionistic Version of Separation Logic

If we define the operations

$$\overset{\mathsf{i}}{\neg}\, p \overset{\text{def}}{=} \mathbf{true} \mathrel{-\!\!*} (\neg\, p)$$
$$p \overset{\mathsf{i}}{\Rightarrow} q \overset{\text{def}}{=} \mathbf{true} \mathrel{-\!\!*} (p \Rightarrow q)$$
$$p \overset{\mathsf{i}}{\Leftrightarrow} q \overset{\text{def}}{=} \mathbf{true} \mathrel{-\!\!*} (p \Leftrightarrow q),$$

then the assertions built from pure assertions and $e \hookrightarrow e'$, using these operations and $\wedge$, $\vee$, $\forall$, $\exists$, $*$, and $\mathrel{-\!\!*}$ form the intuitionistic version of separation logic.

Some Derived Inference Rules

$$\frac{}{q * (q -\!\!* \, p) \Rightarrow p}$$

1. $q * (q -\!\!* \, p) \Rightarrow (q -\!\!* \, p) * q$ $\qquad (p_0 * p_1 \Rightarrow p_1 * p_0)$
2. $(q -\!\!* \, p) \Rightarrow (q -\!\!* \, p)$ $\qquad (p \Rightarrow p)$
3. $(q -\!\!* \, p) * q \Rightarrow p$ $\qquad$ (decurrying, 2)
4. $q * (q -\!\!* \, p) \Rightarrow p$ $\qquad$ (trans impl, 1, 3)

where *transitive implication* is the inference rule

$$\frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r.}$$

$$\frac{}{r \Rightarrow (q \mathrel{-\!\!*} (q * r))}$$

1.	$(r * q) \Rightarrow (q * r)$	$(p_0 * p_1 \Rightarrow p_1 * p_0)$
2.	$r \Rightarrow (q \mathrel{-\!\!*} (q * r))$	(currying, 1)

$$\frac{}{(p \ast r) \Rightarrow (p \ast (q \mathrel{-\!\!\ast} (q \ast r)))}$$

1. $p \Rightarrow p$ $(p \Rightarrow p)$
2. $r \Rightarrow (q \mathrel{-\!\!\ast} (q \ast r))$ (derived above)
3. $(p \ast r) \Rightarrow (p \ast (q \mathrel{-\!\!\ast} (q \ast r)))$ (monotonicity, 1, 2)

$$\frac{p_0 \Rightarrow (q \mathrel{-\!\!*} r) \qquad p_1 \Rightarrow (r \mathrel{-\!\!*} s)}{p_1 * p_0 \Rightarrow (q \mathrel{-\!\!*} s)}$$

1.	$p_1 \Rightarrow p_1$	$(p \Rightarrow p)$
2.	$p_0 \Rightarrow (q \mathrel{-\!\!*} r)$	(assumption)
3.	$p_0 * q \Rightarrow r$	(decurrying, 2)
4.	$p_1 * p_0 * q \Rightarrow p_1 * r$	(monotonicity, 1, 3)
5.	$p_1 \Rightarrow (r \mathrel{-\!\!*} s)$	(assumption)
6.	$p_1 * r \Rightarrow s$	(decurrying, 5)
7.	$p_1 * p_0 * q \Rightarrow s$	(trans impl, 4, 6)
8.	$p_1 * p_0 \Rightarrow (q \mathrel{-\!\!*} s)$	(currying, 7)

$$\frac{p' \Rightarrow p \qquad q \Rightarrow q'}{(p \mathrel{-\!\!*} q) \Rightarrow (p' \mathrel{-\!\!*} q')}.$$

1.	$(p \mathrel{-\!\!*} q) \Rightarrow (p \mathrel{-\!\!*} q)$	$(p \Rightarrow p)$
2.	$p' \Rightarrow p$	(assumption)
3.	$(p \mathrel{-\!\!*} q) * p' \Rightarrow (p \mathrel{-\!\!*} q) * p$	(monotonicity, 1, 2)
4.	$(p \mathrel{-\!\!*} q) * p \Rightarrow q$	(decurrying, 1)
5.	$(p \mathrel{-\!\!*} q) * p' \Rightarrow q$	(trans impl, 3, 4)
6.	$q \Rightarrow q'$	(assumption)
7.	$(p \mathrel{-\!\!*} q) * p' \Rightarrow q'$	(trans impl, 5, 6)
8.	$(p \mathrel{-\!\!*} q) \Rightarrow (p' \mathrel{-\!\!*} q')$	(currying, 7)

Exercise 1

Give a formal proof of the valid assertion

$$\big((\mathsf{x} \mapsto \mathsf{y} \ast \mathsf{x}' \mapsto \mathsf{y}') \ast \mathbf{true}\big) \Rightarrow$$
$$\big(((\mathsf{x} \mapsto \mathsf{y} \ast \mathbf{true}) \wedge (\mathsf{x}' \mapsto \mathsf{y}' \ast \mathbf{true})) \wedge \mathsf{x} \neq \mathsf{x}'\big)$$

from the rules in (2.3) and (2.4), and (some of) the following (derived) inference rules for predicate calculus:

$$p \Rightarrow \mathbf{true} \qquad p \Rightarrow p \qquad p \wedge \mathbf{true} \Rightarrow p$$

$$\frac{p \Rightarrow q \qquad q \Rightarrow r}{p \Rightarrow r} \quad \text{(trans impl)}$$

$$\frac{p \Rightarrow q \qquad p \Rightarrow r}{p \Rightarrow q \wedge r} \quad (\wedge\text{-introduction})$$

Your proof will be easier to read if you write it as a sequence of steps rather than a tree. In the inference rules, you should regard $\ast$ as left associative, e.g.,

$$e_0 \mapsto e_0' \ast e_1 \mapsto e_1' \ast \mathbf{true} \Rightarrow e_0 \neq e_1$$

stands for

$$(e_0 \mapsto e_0' \ast e_1 \mapsto e_1') \ast \mathbf{true} \Rightarrow e_0 \neq e_1.$$

For brevity, you may weaken $\Leftrightarrow$ to $\Rightarrow$ when it is the main operator of an axiom. You may also omit instances of the axiom schema $p \Rightarrow p$ when it is used as a premiss of the monotonicity rule.

Exercise 2

None of the following axiom schemata are sound. For each, given an instance which is not valid, along with a description of a state in which the instance is false.

$$p_0 * p_1 \Rightarrow p_0 \wedge p_1$$

$$p_0 \wedge p_1 \Rightarrow p_0 * p_1$$

$$(p_0 * p_1) \vee q \Rightarrow (p_0 \vee q) * (p_1 \vee q)$$

$$(p_0 \vee q) * (p_1 \vee q) \Rightarrow (p_0 * p_1) \vee q$$

$$(p_0 * q) \wedge (p_1 * q) \Rightarrow (p_0 \wedge p_1) * q$$

$$(p_0 * p_1) \wedge q \Rightarrow (p_0 \wedge q) * (p_1 \wedge q)$$

$$(p_0 \wedge q) * (p_1 \wedge q) \Rightarrow (p_0 * p_1) \wedge q$$

$$\big(\forall x.\ (p_0 * p_1)\big) \Rightarrow (\forall x.\ p_0) * p_1 \quad \text{when } x \text{ not free in } p_1$$

$$(p_0 \Rightarrow p_1) \Rightarrow \big((p_0 * q) \Rightarrow (p_1 * q)\big)$$

$$(p_0 \Rightarrow p_1) \Rightarrow (p_0 -\!\!* \; p_1)$$

$$(p_0 -\!\!* \; p_1) \Rightarrow (p_0 \Rightarrow p_1)$$

AN INTRODUCTION TO

SEPARATION LOGIC

3. Specifications

John C. Reynolds
Carnegie Mellon University
Marktoberdorf August 15, 2008

Hoare Triples

- A *partial correctness specification*

$$\{p\}\ c\ \{q\}$$

  is *valid* iff, starting in any state in which p holds:
  - No execution of c aborts.
  - When some execution of c terminates in a final state, then q holds in the final state.

(We will not consider total correctness in these lectures.)

Examples of Valid Specifications

$$\{x - y > 3\}\ x := x - y\ \{x > 3\}$$

$$\{x + y \geq 17\}\ x := x + 10\ \{x + y \geq 27\}$$

$$\{\mathbf{emp}\}\ x := \mathbf{cons}(1, 2)\ \{x \mapsto 1, 2\}$$

$$\{x \mapsto 1, 2\}\ y := [x]\ \{x \mapsto 1, 2 \wedge y = 1\}$$

$$\{x \mapsto 1, 2 \wedge y = 1\}\ [x + 1] := 3\ \{x \mapsto 1, 3 \wedge y = 1\}$$

$$\{x \mapsto 1, 3 \wedge y = 1\}\ \mathbf{dispose}\ x\ \{x + 1 \mapsto 3 \wedge y = 1\}$$

$$\{x \leq 10\}\ \mathbf{while}\ x \neq 10\ \mathbf{do}\ x := x + 1\ \{x = 10\}$$

$$\{\mathbf{true}\}\ \mathbf{while}\ x \neq 10\ \mathbf{do}\ x := x + 1\ \{x = 10\} \quad (*)$$

$$\{x > 10\}\ \mathbf{while}\ x \neq 10\ \mathbf{do}\ x := x + 1\ \{\mathbf{false}\} \quad (*)$$

$$\{x \mapsto -\ *\ y \mapsto -\}$$

$$\mathbf{if}\ y = x + 1\ \mathbf{then\ skip}$$

$$\quad \mathbf{else\ if}\ x = y + 1\ \mathbf{then}\ x := y$$

$$\quad \mathbf{else}\ (\mathbf{dispose}\ x\,;\ \mathbf{dispose}\ y\,;\ x := \mathbf{cons}(1, 2))$$

$$\{x \mapsto -, -\}$$

(All except the examples marked $(*)$ are also valid total specifications.)

Inference Rules and Proofs

- An *inference rule* for Hoare logic consists of zero or more *premisses* (either specifications or assertions) and a single *conclusion* (a specification), separated by a horizontal line:

$$\frac{\mathcal{P}_1 \qquad \cdots \qquad \mathcal{P}_n}{\mathcal{C}}$$

- The premisses and conclusion are schemata, i.e., they may contain *metavariables*, each of which ranges over some set of phrases, such as expressions, commands, or assertions.
- An instance of an inference rule is obtained by replacing each metavariable by a phrase in its range. These replacements must satisfy the *side-conditions* (if any) of the rule. (Since this is replacement of metavariables rather than substitution for variables, there is never any renaming.)
- A *formal proof* in Hoare logic is a sequence of assertions and/or specifications, each of which is either a valid assertion or the conclusion of some instance of a sound inference rule whose premisses occur earlier in the sequence,

Hoare's Inference Rules for Specifications: Assignment (AS)

$$\frac{}{\{p/v \to e\}\ v := e\ \{p\}}$$

Instances

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

$$\frac{}{\{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}$$

Sequential Composition (SQ)

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\,;\,c_2\ \{r\}}$$

An Instance

$$\frac{\begin{array}{r}\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \{2 \times \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\end{array}}{\begin{array}{r}\{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k}+1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k}+1\,;\,\mathsf{y} := 2 \times \mathsf{y} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}\end{array}}$$

Strengthening Precedent (SP)

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\ c\ \{r\}}$$

An Instance

$$\frac{\begin{array}{l} \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n} \Rightarrow 2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n} \\ \quad \{2 \times \mathsf{y} = 2^{\mathsf{k}+1} \wedge \mathsf{k} + 1 \leq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y} \\ \hfill \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}}{\begin{array}{l} \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y} \\ \hfill \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \end{array}}$$

Since they are applicable to arbitrary commands, the rules (SP) and (WC) (to be introduced later) are called *structural rules*. One premiss of each of these rules is an assertion, which is called a *verification condition* (VC). The verification conditions are used to introduce mathematical facts about data types into proofs of specifications.

We will usually omit formal proofs of verification conditions.

Partial Correctness of **while** (WH)

$$\frac{\{i \wedge b\}\ c\ \{i\}}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{i \wedge \neg b\}}$$

Here i is the *invariant*.

An Instance

$$\frac{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \mathsf{k} \neq \mathsf{n}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\}\end{array}}$$

Weakening Consequent (WC)

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\ c\ \{r\}}$$

An Instance

$$\frac{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n}\} \\ \\ \mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n} \wedge \neg\, \mathsf{k} \neq \mathsf{n} \Rightarrow \mathsf{y} = 2^{\mathsf{n}}\end{array}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\ \textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

A Proof

1. $y = 2^k \wedge k \leq n \wedge k \neq n \Rightarrow 2 \times y = 2^{k+1} \wedge k + 1 \leq n$ (VC)
2. $\{2 \times y = 2^{k+1} \wedge k + 1 \leq n\}\, k := k + 1$
   $\{2 \times y = 2^k \wedge k \leq n\}$ (AS)
3. $\{2 \times y = 2^k \wedge k \leq n\}\; y := 2 \times y\; \{y = 2^k \wedge k \leq n\}$ (AS)
4. $\{2 \times y = 2^{k+1} \wedge k + 1 \leq n\}\, k := k + 1 \,;\, y := 2 \times y$
   $\{y = 2^k \wedge k \leq n\}$ (SQ 2,3)
5. $\{y = 2^k \wedge k \leq n \wedge k \neq n\}\, k := k + 1 \,;\, y := 2 \times y$
   $\{y = 2^k \wedge k \leq n\}$ (SP 1,4)
6. $\{y = 2^k \wedge k \leq n\}$ (WH 5)
   $\textbf{while}\; k \neq n\; \textbf{do}\; (k := k + 1 \,;\, y := 2 \times y)$
   $\{y = 2^k \wedge k \leq n \wedge \neg\, k \neq n\}$
7. $y = 2^k \wedge k \leq n \wedge \neg\, k \neq n \Rightarrow y = 2^n$ (VC)
8. $\{y = 2^k \wedge k \leq n\}$ (WC 6,7)
   $\textbf{while}\; k \neq n\; \textbf{do}\; (k := k + 1 \,;\, y := 2 \times y)$
   $\{y = 2^n\}$

skip (SK)

$$\frac{}{\{p\}\ \mathbf{skip}\ \{p\}}$$

An Instance

$$\{\mathsf{y} = 2^{\mathsf{k}} \land \neg\, \mathsf{odd}(\mathsf{k})\}\ \mathbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \land \neg\, \mathsf{odd}(\mathsf{k})\}$$

Conditional (CD)

$$\frac{\{p \wedge b\}\ c_1\ \{q\} \qquad \{p \wedge \neg\ b\}\ c_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

An Instance

$$\frac{\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{odd(k)}\}\ \mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathsf{odd(k)}\} \qquad \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathsf{odd(k)}\}\ \textbf{skip}\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathsf{odd(k)}\}}{\begin{array}{l}\{\mathsf{y} = 2^{\mathsf{k}}\} \\ \textbf{if}\ \mathsf{odd(k)}\ \textbf{then}\ \mathsf{k} := \mathsf{k} + 1 ; \mathsf{y} := 2 \times \mathsf{y}\ \textbf{else}\ \textbf{skip} \\ \{\mathsf{y} = 2^{\mathsf{k}} \wedge \neg\,\mathsf{odd(k)}\}\end{array}}$$

Variable Declaration (DC)

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ \mathbf{newvar}\ v\ \mathbf{in}\ c\ \{q\}}$$

when v does not occur free in p or q.

An Instance

$$\frac{\begin{array}{l}\{1 = 2^0 \wedge 0 \leq \mathsf{n}\} \\ \mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ; \\ \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}{\begin{array}{l}\{1 = 2^0 \wedge 0 \leq \mathsf{n}\} \\ \mathbf{newvar}\ \mathsf{k}\ \mathbf{in} \\ \quad \big(\mathsf{k} := 0\ ;\ \mathsf{y} := 1\ ; \\ \quad \mathbf{while}\ \mathsf{k} \neq \mathsf{n}\ \mathbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})\big) \\ \{\mathsf{y} = 2^{\mathsf{n}}\}\end{array}}$$

Here the requirement on the declared variable v formalizes the concept of *locality*, i.e., that the value of v when c begins execution has no effect on this execution, and that the value of v when c finishes execution has no effect on the rest of the program.

Notice that locality is context-dependent: In

$$\{\mathbf{true}\}\ \mathsf{t} := \mathsf{x} + \mathsf{y}\ ;\ \mathsf{y} := \mathsf{t} \times 2\ \{\mathsf{y} = (\mathsf{x} + \mathsf{y}) \times 2\},$$

t is local, and can be declared at the beginning of the command being specified, but in

$$\{\mathbf{true}\}\ \mathsf{t} := \mathsf{x} + \mathsf{y}\ ;\ \mathsf{y} := \mathsf{t} \times 2\ \{\mathsf{y} = (\mathsf{x} + \mathsf{y}) \times 2 \wedge \mathsf{t} = (\mathsf{x} + \mathsf{y})\},$$

t is not local, and cannot be declared.

Why Annotations Are Needed

Without annotations, it is not straightforward to construct a proof of a specification from the specification itself. For example, if we try to use the rule for sequential composition,

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ c_2\ \{r\},}$$

to obtain the main step of a proof of the specification

$$\{\mathsf{n} \geq 0\}$$
$$(\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ;$$
$$\textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y})$$
$$\{\mathsf{y} = 2^{\mathsf{n}}\},$$

there is no indication of what assertion should replace the metavariable q.

Why Annotations Are Needed (continued)

But if we change the rule to

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\ ;\ \underline{\{q\}}\ c_2\ \{r\}},$$

then the new rule requires the annotation q to occur in the conclusion:

$$\begin{array}{l}
\{\mathsf{n} \geq 0\} \\
(\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ; \\
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do}\ (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\
\{\mathsf{y} = 2^{\mathsf{n}}\}.
\end{array}$$

Then, once q is determined, the premisses must be

$$\begin{array}{l}
\{\mathsf{n} \geq 0\} \\
(\mathsf{k} := 0\ ;\ \mathsf{y} := 1)\ ; \\
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}
\end{array}
\qquad \text{and} \qquad
\begin{array}{l}
\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\} \\
\textbf{while}\ \mathsf{k} \neq \mathsf{n}\ \textbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{y} := 2 \times \mathsf{y}) \\
\{\mathsf{y} = 2^{\mathsf{n}}\}.
\end{array}$$

The basic trick is to add annotations to the conclusions of the inference rules so that the conclusion of each rule completely determines its premisses.

Why Do We Ever Need Intermediate Assertions?

1. **while** commands and calls of recursive procedures do not always have weakest preconditions that can be expressed in our assertion language.
2. Certain inference rules, such as the frame rule, do not fit well into the framework of weakest assertions.
3. Intermediate assertions are often needed to simplify verification conditions.

More Structural Inference Rules

Disjunction (DISJ)

$$\frac{\{p_1\}\ c\ \{q\} \qquad \{p_2\}\ c\ \{q\}}{\{p_1 \vee p_2\}\ c\ \{q\}}$$

For example, from

$$\begin{array}{l}
\{\mathsf{a} - 1 \leq \mathsf{b}\} \\
\mathsf{s} := 0\ ;\ \mathsf{k} := \mathsf{a} - 1\ ; \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{k}} \mathsf{i} \wedge \mathsf{k} \leq \mathsf{b}\} \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k}) \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}
\end{array}$$

$$\begin{array}{l}
\{\mathsf{a} - 1 \geq \mathsf{b}\} \\
\mathsf{s} := 0\ ;\ \mathsf{k} := \mathsf{a} - 1\ ; \\
\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b} \wedge \mathsf{k} \geq \mathsf{b}\} \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k}) \\
\{\mathsf{s} = 0 \wedge \mathsf{a} - 1 \geq \mathsf{b}\} \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}.
\end{array}$$

we can obtain the main step in

$$\begin{array}{l}
\{\mathbf{true}\} \\
\{\mathsf{a} - 1 \leq \mathsf{b} \vee \mathsf{a} - 1 \geq \mathsf{b}\} \\
\mathsf{s} := 0\ ;\ \mathsf{k} := \mathsf{a} - 1\ ; \\
\mathbf{while}\ \mathsf{k} < \mathsf{b}\ \mathbf{do} \\
\quad (\mathsf{k} := \mathsf{k} + 1\ ;\ \mathsf{s} := \mathsf{s} + \mathsf{k}) \\
\{\mathsf{s} = \sum_{\mathsf{i}=\mathsf{a}}^{\mathsf{b}} \mathsf{i}\}.
\end{array}$$

Conjunction (CONJ)

$$\frac{\{p\}\ c\ \{q_1\} \qquad \{p\}\ c\ \{q_2\}}{\{p\}\ c\ \{q_1 \wedge q_2\}}$$

Existential Quantification (EQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\}},$$

where v is not free in c.

Substitution (SUB)

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/v_1 \to e_1,\ \ldots,\ v_n \to e_n,}$$

where $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

The restrictions on this rule are needed to avoid aliasing.

For example, in

$$\{\mathsf{x} = \mathsf{y}\}\ \mathsf{x} := \mathsf{x} + \mathsf{y}\ \{\mathsf{x} = 2 \times \mathsf{y}\},$$

one can substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{w} - 1$ to infer

$$\{\mathsf{z} = 2 \times \mathsf{w} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{w} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{w} - 1)\}.$$

But one cannot substitute $\mathsf{x} \to \mathsf{z}$, $\mathsf{y} \to 2 \times \mathsf{z} - 1$ to infer the invalid

$$\{\mathsf{z} = 2 \times \mathsf{z} - 1\}\ \mathsf{z} := \mathsf{z} + 2 \times \mathsf{z} - 1\ \{\mathsf{z} = 2 \times (2 \times \mathsf{z} - 1)\}.$$

The Frame Rule (O'Hearn) (FR)

$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$

where no variable occurring free in r is modified by c.

An Instance

$$\frac{\{\mathbf{list}\,\alpha\,\mathsf{i}\}\ \text{“Reverse List”}\ \{\mathbf{list}\,\alpha^\dagger\,\mathsf{j}\}}{\{\mathbf{list}\,\alpha\,\mathsf{i}\ *\ \mathbf{list}\,\gamma\,\mathsf{x}\}\ \text{“Reverse List”}\ \{\mathbf{list}\,\alpha^\dagger\,\mathsf{j}\ *\ \mathbf{list}\,\gamma\,\mathsf{x}\}},$$

(assuming “Reverse List” does not modify x or γ).

The Delicacy of the Frame Rule

Suppose

$$\{\mathbf{emp}\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathbf{emp}\}.$$

Then the frame rule would give

$$\{\mathbf{emp} * \mathsf{x} \mapsto 10\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathbf{emp} * \mathsf{x} \mapsto 10\},$$

and therefore

$$\{\mathsf{x} \mapsto 10\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathsf{x} \mapsto 10\},$$

which is patently false.

Why the Frame Rule is Sound

We define:

If, starting in the state s, h, no execution of a command c aborts, then c is *safe at* s, h.

If, starting in the state s, h, every execution of c terminates without aborting, then c *must terminate normally* at s, h.

Then our programming language satisfies:

Safety Monotonicity If $\widehat{h} \subseteq h$ and c is safe at $s, \widehat{h}$, then c is safe at s, h. If $\widehat{h} \subseteq h$ and c must terminate normally at $s, \widehat{h}$, then c must terminate normally at s, h.

The Frame Property If $\widehat{h} \subseteq h$, c is safe at $s, \widehat{h}$, and some execution of c starting at s, h terminates normally in the state s', h', then $h - \widehat{h} \subseteq h'$ and some execution of c starting at $s, \widehat{h}$, terminates normally in the state $s', h' - (h - \widehat{h})$.

Then:

Proposition 5 *If the programming language satisfies safety monotonicity and the frame property, then the frame rule is sound for both partial and total correctness.*

Annotating (FR) and (EQ)

$$
\begin{array}{l}
\{\exists \mathsf{j}.\ \mathsf{x} \mapsto -, \mathsf{j} \ast \mathsf{list}\,\alpha\,\mathsf{j}\} \\
\quad \left.\left.\begin{array}{l}
\{\mathsf{x} \mapsto -\} \\
[\mathsf{x}] := \mathsf{a} \\
\{\mathsf{x} \mapsto \mathsf{a}\}
\end{array}\right\} \ast \mathsf{x} + 1 \mapsto \mathsf{j} \ast \mathsf{list}\,\alpha\,\mathsf{j} \right\} \exists \mathsf{j} \\
\{\exists \mathsf{j}.\ \mathsf{x} \mapsto \mathsf{a}, \mathsf{j} \ast \mathsf{list}\,\alpha\,\mathsf{j}\}
\end{array}
$$

Inference Rules for Mutation

The local form (MUL):

$$\overline{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

The global form (MUG):

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}.}$$

The backward-reasoning form (MUBR):

$$\overline{\{(e \mapsto -) \ast ((e \mapsto e') -\!\!\ast\ p)\}\ [e] := e'\ \{p\}.}$$

The local form (MUL):

$$\overline{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}}.$$

The global form (MUG):

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}}.$$

One can derive (MUG) from (MUL) by using the frame rule:

$$\begin{array}{l}
\{(e \mapsto -) \ast r\} \\
\qquad \left.\begin{array}{l}
\{e \mapsto -\} \\
[e] := e' \\
\{e \mapsto e'\}
\end{array}\right\} \ast r \\
\{(e \mapsto e') \ast r\},
\end{array}$$

The local form (MUL):

$$\overline{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

The global form (MUG):

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}.}$$

while to go in the opposite direction it is only necessary to take r to be $\mathbf{emp}$:

$$\begin{array}{l}\{e \mapsto -\}\\ \{(e \mapsto -) \ast \mathbf{emp}\}\\ [e] := e'\\ \{(e \mapsto e') \ast \mathbf{emp}\}\\ \{e \mapsto e'\}.\end{array}$$

The global form (MUG):

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}.}$$

The backward-reasoning form (MUBR):

$$\overline{\{(e \mapsto -) \ast ((e \mapsto e') -\!\!\ast\, p)\}\ [e] := e'\ \{p\}.}$$

One can derive (MUBR) from (MUG) by taking r to be $(e \mapsto e') -\!\!\ast\, p$ and using the law $q \ast (q -\!\!\ast\, p) \Rightarrow p$:

$$\begin{array}{l}
\{(e \mapsto -) \ast ((e \mapsto e') -\!\!\ast\, p)\} \\
[e] := e' \\
\{(e \mapsto e') \ast ((e \mapsto e') -\!\!\ast\, p)\} \\
\{p\}.
\end{array}$$

The global form (MUG):

$$\overline{\{(e \mapsto -) \ast r\}\ [e] := e'\ \{(e \mapsto e') \ast r\}.}$$

The backward-reasoning form (MUBR):

$$\overline{\{(e \mapsto -) \ast ((e \mapsto e') -\!\!\ast\ p)\}\ [e] := e'\ \{p\}.}$$

One can go in the opposite direction by taking p to be $(e \mapsto e') \ast r$ and using $(p \ast r) \Rightarrow (p \ast (q -\!\!\ast (q \ast r)))$:

$$\begin{aligned}
&\{(e \mapsto -) \ast r\} \\
&\{(e \mapsto -) \ast ((e \mapsto e') -\!\!\ast ((e \mapsto e') \ast r))\} \\
&[e] := e' \\
&\{(e \mapsto e') \ast r\}.
\end{aligned}$$

Inference Rules for Deallocation

The local form (DISL):

$$\overline{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}.}$$

The global (and backward-reasoning) form (DISG):

$$\overline{\{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}.}$$

One can derive (DISG) from (DISL) by using (FR); one can go in the opposite direction by taking r to be $\mathbf{emp}$.

Inference Rules for Allocation and Lookup

These are *generalized assignment commands*, but they don't obey the assignment rule, e.g.

$$\{\mathbf{cons}(1,2) = \mathbf{cons}(1,2)\}\ \mathsf{x} := \mathbf{cons}(1,2)\ \{\mathsf{x} = \mathsf{x}\}$$

$$\uparrow$$

syntactically illegal

$$\downarrow$$

$$\{[\mathsf{y}] = [\mathsf{y}]\}\ \mathsf{x} := [\mathsf{y}]\ \{\mathsf{x} = \mathsf{x}\}$$

Inference Rules for Nonoverwriting Allocation

We abbreviate the sequence $e_1, \ldots, e_n$ of expressions by $\overline{e}$:

- The local form (CONSNOL)

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}\}},$$

  where $v \notin \mathsf{FV}(\overline{e})$.

- The global form (CONSNOG)

$$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e})\ *\ r\}},$$

  where $v \notin \mathsf{FV}(\overline{e}, r)$.

Again, one can derive the global form from the local by using the frame rule, and the local from the global by taking r to be $\mathbf{emp}$.

Inference Rules for General Allocation

- The local form (CONSL)

$$\overline{\{v = v' \wedge \mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}'\},}$$

  where v' is distinct from v, and $\overline{e}'$ denotes $\overline{e}/v \to v'$ (i.e., each e_i' denotes $e_i/v \to v'$).

- The global form (CONSG)

$$\overline{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\},}$$

  where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

- The backward-reasoning form (CONSBR)

$$\overline{\{\forall v''.\ (v'' \mapsto \overline{e}) -\!\!* \ p''\}\ v := \mathbf{cons}(\overline{e})\ \{p\},}$$

  where v'' is distinct from v, $v'' \notin \mathsf{FV}(\overline{e}, p)$, and p'' denotes $p/v \to v''$.

An Instance of (CONSG)

$$\overline{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{\exists v'.\ (v \mapsto \overline{e}')\ *\ r'\}},$$

where v' is distinct from v, $v' \notin \mathsf{FV}(\overline{e}, r)$, $\overline{e}'$ denotes $\overline{e}/v \to v'$, and r' denotes $r/v \to v'$.

An Instance:

$$\{\mathsf{list}\,\alpha\,\mathsf{i}\}\ \mathsf{i} := \mathbf{cons}(3, \mathsf{i})\ \{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathsf{list}\,\alpha\,\mathsf{j}\}.$$

Inference Rules for Nonoverwriting Lookup

- The local nonoverwriting form (LKNOL)

$$\overline{\{e \mapsto v''\}\; v := [e]\; \{v = v'' \wedge (e \mapsto v)\},}$$

  where $v \notin \mathsf{FV}(e)$.

- The global nonoverwriting form (LKNOG)

$$\overline{\{\exists v''.\, (e \mapsto v'') \, * \, p''\}\; v := [e]\; \{(e \mapsto v) \, * \, p\},}$$

  where $v \notin \mathsf{FV}(e)$, $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \rightarrow v''$.

In (LKNOG):

$$\overline{\{\exists v''.\ (e \mapsto v'')\ *\ p''\}\ v := [e]\ \{(e \mapsto v)\ *\ p\},}$$

where $v \notin \mathsf{FV}(e)$, $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

there is no restriction preventing v'' from being the same variable as v. Thus, as a special case,

$$\overline{\{\exists v.\ (e \mapsto v)\ *\ p\}\ v := [e]\ \{(e \mapsto v)\ *\ p\},}$$

where $v \notin \mathsf{FV}(e)$. For example, if we take

v to be j
e to be $\mathsf{i} + 1$
p to be $\mathsf{i} \mapsto 3\ *\ \mathbf{list}\ \alpha\ \mathsf{j}$,

(and remember $\mathsf{i} \mapsto 3, \mathsf{j}$ abbreviates $(\mathsf{i} \mapsto 3) * (\mathsf{i} + 1 \mapsto \mathsf{j})$), then we obtain the instance

$$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathbf{list}\ \alpha\ \mathsf{j}\}\ \mathsf{j} := [\mathsf{i} + 1]\ \{\mathsf{i} \mapsto 3, \mathsf{j}\ *\ \mathbf{list}\ \alpha\ \mathsf{j}\}.$$

Inference Rules for General Lookup

- The local form (LKL)

$$\overline{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}$$

where v, v', and v'' are distinct, and e' denotes $e/v \to v'$.

- The global form (LKG)

$$\overline{\begin{array}{l}\{\exists v''.\ (e \mapsto v'') * (r/v' \to v)\}\ v := [e] \\ \qquad\qquad\qquad \{\exists v'.\ (e' \mapsto v) * (r/v'' \to v)\},\end{array}}$$

where v, v', and v'' are distinct, v', $v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \to v'$.

- The first backward-reasoning form (LKBR1)

$$\overline{\{\exists v''.\ (e \mapsto v'') * ((e \mapsto v'') -\!\!* p'')\}\ v := [e]\ \{p\},}$$

where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

- The second backward-reasoning form (LKBR2)

$$\overline{\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\ v := [e]\ \{p\},}$$

where $v'' \notin \mathsf{FV}(e) \cup (\mathsf{FV}(p) - \{v\})$, and p'' denotes $p/v \to v''$.

The Soundness of the Local Rule (LKL)

$$\overline{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v)\},}$$

where v, v', and v'' are distinct, and e' denotes $e/v \to v'$.

Suppose that the precondition holds in the state s_0, h, i.e., that

$$s_0, h \vDash v = v' \wedge (e \mapsto v'').$$

Then $s_0\, v = s_0\, v'$ and $h = [\, [\![e]\!]_{\mathsf{exp}} s_0 \colon s_0\, v''\,]$.

Starting in the state s_0, h, the execution of $v := [e]$ will not abort (since $[\![e]\!]_{\mathsf{exp}} s_0 \in \mathrm{dom}\, h$), and will terminate with the store

$$s_1 = [\, s_0 \mid v \colon s_0\, v''\,]$$

and the unchanged heap h. To see that this state satisfies the postcondition, we note that $s_1\, v = s_0\, v'' = s_1\, v''$ and, since e' does not contain v, $[\![e']\!]_{\mathsf{exp}} s_1 = [\![e']\!]_{\mathsf{exp}} s_0$. Then applying the substitution law for assertions, with

$$\widehat{s} = [\, s_0 \mid v \colon s_0\, v'\,] = [\, s_0 \mid v \colon s_0\, v\,] = s_0,$$

we obtain $[\![e']\!]_{\mathsf{exp}} s_0 = [\![e]\!]_{\mathsf{exp}} s_0$. Thus

$$h = [\, [\![e']\!]_{\mathsf{exp}} s_1 \colon s_1\, v\,] \quad \text{and} \quad s_1, h \vDash v = v'' \wedge (e' \mapsto v).$$

An Instance of (LKG)

$$\overline{\{\exists v''.\, (e \mapsto v'') * (r/v' \to v)\}\ v := [e]\ \{\exists v'.\, (e' \mapsto v) * (r/v'' \to v)\},}$$

where v, v', and v'' are distinct, v', $v'' \notin \mathsf{FV}(e)$, $v \notin \mathsf{FV}(r)$, and e' denotes $e/v \to v'$.

As an example of an instance, if we take

v to be j		e to be $\mathsf{j}+1$	
v' to be m		r to be $\mathsf{i}+1 \mapsto \mathsf{m} * \mathsf{k}+1 \mapsto \mathbf{nil}$,	
v'' to be k			

then we obtain (using the commutivity of $*$)

$$\{\exists \mathsf{k}.\ \mathsf{i}+1 \mapsto \mathsf{j} * \mathsf{j}+1 \mapsto \mathsf{k} * \mathsf{k}+1 \mapsto \mathbf{nil}\}$$
$$\mathsf{j} := [\mathsf{j}+1]$$
$$\{\exists \mathsf{m}.\ \mathsf{i}+1 \mapsto \mathsf{m} * \mathsf{m}+1 \mapsto \mathsf{j} * \mathsf{j}+1 \mapsto \mathbf{nil}\}.$$

A Final Example

$\{\mathbf{emp}\}$

$\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a})\ ;$ (CONSNOL)

$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\}$ i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{a}\}$

$\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b})\ ;$ (CONSNOG)

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{a} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{b}\}$

$(p/v \rightarrow e \Rightarrow \exists v.\ p)$

$\{(\mathsf{x} \mapsto \mathsf{a}, -) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast (\exists \mathsf{a}.\ \mathsf{x} + 1 \mapsto \mathsf{a}) \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{b}\}$

$[\mathsf{x} + 1] := \mathsf{y} - \mathsf{x}\ ;$ (MUG)

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{b}\}$

$(p/v \rightarrow e \Rightarrow \exists v.\ p)$

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, -)\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast (\exists \mathsf{b}.\ \mathsf{y} + 1 \mapsto \mathsf{b})\}$

$[\mathsf{y} + 1] := \mathsf{x} - \mathsf{y}\ ;$ (MUG)

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto \mathsf{x} - \mathsf{y}\}$

$(x - y = -(y - x))$

$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \ast (\mathsf{y} \mapsto \mathsf{b}, -(\mathsf{y} - \mathsf{x}))\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{y} - \mathsf{x} \ast \mathsf{y} \mapsto \mathsf{b} \ast \mathsf{y} + 1 \mapsto -(\mathsf{y} - \mathsf{x})\}$

$(p/v \rightarrow e \Rightarrow \exists v.\ p)$

$\{\exists \mathsf{o}.\ (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \ast (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b}, -\mathsf{o})\}$

i.e., $\{\mathsf{x} \mapsto \mathsf{a} \ast \mathsf{x} + 1 \mapsto \mathsf{o} \ast \mathsf{x} + \mathsf{o} \mapsto \mathsf{b} \ast \mathsf{x} + \mathsf{o} + 1 \mapsto -\mathsf{o}\}$

Exercise 3

Fill in the postconditions in

$$\{(e_1 \mapsto -) \ast (e_2 \mapsto -)\}\ [e_1] := e_1' \,;\, [e_2] := e_2'\ \{?\}$$

$$\{(e_1 \mapsto -) \wedge (e_2 \mapsto -)\}\ [e_1] := e_1' \,;\, [e_2] := e_2'\ \{?\}.$$

to give two sound inference rules describing a sequence of two mutations. Your postconditions should be as strong as possible.

Give a derivation of each of these inference rules, exhibited as an annotated specification.

AN INTRODUCTION TO

SEPARATION LOGIC

4. Lists and List Segments

John C. Reynolds
Carnegie Mellon University
Marktoberdorf August 15, 2008

Notation for Sequences

When α and β are sequences, we write

- ϵ for the empty sequence.
- $[a]$ for the single-element sequence containing a. (We will omit the brackets when a is not a sequence.)
- $\alpha{\cdot}\beta$ for the composition of α followed by β.
- $\alpha^\dagger$ for the reflection of α.
- $\#\alpha$ for the length of α.
- α_i for the ith component of α.

Some Laws for Sequences

$$\alpha{\cdot}\epsilon = \alpha \qquad \epsilon{\cdot}\alpha = \alpha \qquad (\alpha{\cdot}\beta){\cdot}\gamma = \alpha{\cdot}(\beta{\cdot}\gamma)$$

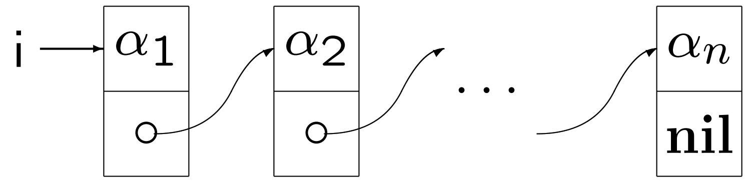
$$\epsilon^\dagger = \epsilon \qquad [a]^\dagger = [a] \qquad (\alpha{\cdot}\beta)^\dagger = \beta^\dagger{\cdot}\alpha^\dagger$$

$$\#\epsilon = 0 \qquad \#[a] = 1 \qquad \#(\alpha{\cdot}\beta) = (\#\alpha) + (\#\beta)$$

$$\alpha = \epsilon \vee \exists a, \alpha'.\, \alpha = [a]{\cdot}\alpha' \qquad \alpha = \epsilon \vee \exists \alpha', a.\, \alpha = \alpha'{\cdot}[a].$$

Singly-linked Lists

list α i:



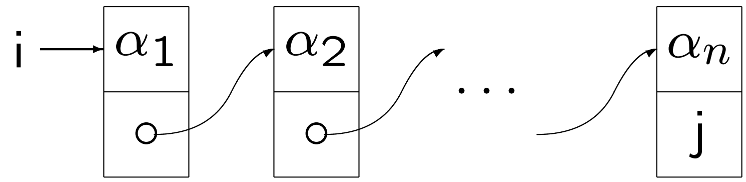
is defined by induction on the length of the sequence α (i.e., by structural induction on α):

$$\mathsf{list}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil}$$

$$\mathsf{list}\ (\mathsf{a}\cdot\alpha)\ \mathsf{i} \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathsf{list}\ \alpha\ \mathsf{j}.$$

Singly-linked List Segments

lseg $\alpha\,(\mathsf{i},\mathsf{j})$:



is defined by

$$\text{lseg}\ \epsilon\,(\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\text{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k}) \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j} \ * \ \text{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}).$$

Properties

$$\text{lseg}\ \mathsf{a}\,(\mathsf{i},\mathsf{j}) \Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j}$$

$$\text{lseg}\ \alpha{\cdot}\beta\,(\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \text{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}) \ * \ \text{lseg}\ \beta\,(\mathsf{j},\mathsf{k})$$

$$\text{lseg}\ \alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \text{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}) \ * \ \mathsf{j} \mapsto \mathsf{b},\mathsf{k}$$

$$\text{list}\ \alpha\ \mathsf{i} \Leftrightarrow \text{lseg}\ \alpha\,(\mathsf{i},\mathbf{nil}).$$

Emptyness Conditions

For lists, one can derive a law that shows clearly when a list represents the empty sequence:

$$\mathsf{list}\ \alpha\ \mathsf{i} \Rightarrow (\mathsf{i} = \mathbf{nil} \Leftrightarrow \alpha = \epsilon).$$

For list segments, however, the situation is more complex. One can derive

$$\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \Rightarrow (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil}))$$
$$\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \Rightarrow (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon).$$

But these formulas do not say whether α is empty when $\mathsf{i} = \mathsf{j} \neq \mathbf{nil}$.

Nontouching List Segments

When

$$\mathsf{lseg}\; a_1 \cdots a_n\, (\mathsf{i}_0, \mathsf{i}_n),$$

we have

$$\exists \mathsf{i}_1, \ldots \mathsf{i}_{n-1}.$$
$$(\mathsf{i}_0 \mapsto a_1, \mathsf{i}_1) \;*\; (\mathsf{i}_1 \mapsto a_2, \mathsf{i}_2) \;*\; \cdots \;*\; (\mathsf{i}_{n-1} \mapsto a_n, \mathsf{i}_n).$$

Thus i_0, . . . , i_{n-1} are distinct, but i_n is not constrained, and may equal any of the i_0, . . . , i_{n-1}. In this case, we say that the list segment is *touching*.

We can define nontouching list segments inductively by:

$$\mathsf{ntlseg}\; \epsilon\, (\mathsf{i}, \mathsf{j}) \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{ntlseg}\; \mathsf{a}{\cdot}\alpha\, (\mathsf{i}, \mathsf{k}) \stackrel{\text{def}}{=} \mathsf{i} \neq \mathsf{k} \wedge \mathsf{i}{+}1 \neq \mathsf{k} \wedge (\exists \mathsf{j}.\; \mathsf{i} {\mapsto} \mathsf{a}, \mathsf{j} * \mathsf{ntlseg}\; \alpha\, (\mathsf{j}, \mathsf{k})),$$

or equivalently, we can define them in terms of lseg:

$$\mathsf{ntlseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \stackrel{\text{def}}{=} \mathsf{lseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \wedge \neg\, \mathsf{j} \hookrightarrow -.$$

The obvious advantage of knowing that a list segment is nontouching is that it is easy to test whether it is empty:

$$\mathsf{ntlseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \Rightarrow (\alpha = \epsilon \Leftrightarrow \mathsf{i} = \mathsf{j}).$$

Fortunately, there are common situations where list segments must be nontouching:

$$\mathsf{list}\; \alpha\; \mathsf{i} \Rightarrow \mathsf{ntlseg}\; \alpha\, (\mathsf{i}, \mathbf{nil})$$
$$\mathsf{lseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \;*\; \mathsf{list}\; \beta\; \mathsf{j} \Rightarrow \mathsf{ntlseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \;*\; \mathsf{list}\; \beta\; \mathsf{j}$$
$$\mathsf{lseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \;*\; \mathsf{j} \hookrightarrow - \Rightarrow \mathsf{ntlseg}\; \alpha\, (\mathsf{i}, \mathsf{j}) \;*\; \mathsf{j} \hookrightarrow -.$$

Preciseness of List Assertions

The assertions

$$\mathsf{list}\ \alpha\ \mathsf{i} \qquad \mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}) \qquad \mathsf{ntlseq}\ \alpha\ (\mathsf{i}, \mathsf{j})$$

are all precise.. On the other hand, although

$$\exists \alpha.\ \mathsf{list}\ \alpha\ \mathsf{i} \qquad \exists \alpha.\ \mathsf{ntlseq}\ \alpha\ (\mathsf{i}, \mathsf{j})$$

are precise,

$$\exists \alpha.\ \mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})$$

is not precise.

Insertion at the Beginning of a List Segment

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\}$$

$$\mathsf{k} := \mathbf{cons}(\mathsf{a},\mathsf{i})\ ; \qquad \text{(CONSNOG)}$$

$$\{\mathsf{k} \mapsto \mathsf{a},\mathsf{i}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\}$$

$$\{\exists \mathsf{i}.\ \mathsf{k} \mapsto \mathsf{a},\mathsf{i}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\}$$

$$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{k},\mathsf{j})\}$$

$$\mathsf{i} := \mathsf{k} \qquad \text{(AS)}$$

$$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{j})\},$$

or, more concisely:

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{k})\}$$

$$\mathsf{i} := \mathbf{cons}(\mathsf{a},\mathsf{i})\ ; \qquad \text{(CONSG)}$$

$$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{lseg}\ \alpha\ (\mathsf{j},\mathsf{k})\}$$

$$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{k})\}.$$

Insertion at the End of a List Segment

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\}$

$\mathsf{l} := \mathbf{cons}(\mathsf{b},\mathsf{k})\ ;$ (CONSNOG)

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto -\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$

$[\mathsf{j}+1] := \mathsf{l}$ (MUG)

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\ (\mathsf{i},\mathsf{l})\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\}$

$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}{\cdot}\mathsf{b}\ (\mathsf{i},\mathsf{k})\}.$

Deletion at the Beginning of a List Segment

$\{\mathsf{lseg}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\}$

$\{\exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\{\exists \mathsf{j}.\ \mathsf{i}+1 \mapsto \mathsf{j} \ast (\mathsf{i} \mapsto \mathsf{a} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}$

$\mathsf{j} := [\mathsf{i}+1]\ ;$ (LKNOG)

$\{\mathsf{i}+1 \mapsto \mathsf{j} \ast (\mathsf{i} \mapsto \mathsf{a} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}$

$\{\mathsf{i} \mapsto \mathsf{a} \ast (\mathsf{i}+1 \mapsto \mathsf{j} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k}))\}$

dispose $\mathsf{i}\ ;$ (DISG)

$\{\mathsf{i}+1 \mapsto \mathsf{j} \ast \mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

dispose $\mathsf{i}+1\ ;$ (DISG)

$\{\mathsf{lseg}\ \alpha\,(\mathsf{j},\mathsf{k})\}$

$\mathsf{i} := \mathsf{j}$ (AS)

$\{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{k})\}.$

Deletion at the End of a List Segment

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\}$$

$$[\mathsf{j}+1] := \mathsf{l}\,; \qquad \text{(MUG)}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{k} \mapsto \mathsf{b},\mathsf{l}\}$$

$$\mathbf{dispose}\ \mathsf{k}\,; \qquad \text{(DISG)}$$

$$\mathbf{dispose}\ \mathsf{k}+1 \qquad \text{(DISG)}$$

$$\{\mathsf{lseg}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\}$$

$$\{\mathsf{lseg}\ \alpha{\cdot}\mathsf{a}\ (\mathsf{i},\mathsf{l})\}.$$

A Cyclic Buffer

$$\exists \beta.\ (\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j}, \mathsf{i})) \wedge \mathsf{m} = \#\alpha \wedge \mathsf{n} = \#\alpha + \#\beta$$

When $\mathsf{i} = \mathsf{j}$, the buffer is either empty ($\#\alpha = 0 \wedge \mathsf{m} = 0$) or full ($\#\beta = 0 \wedge \mathsf{m} = \mathsf{n}$).

Simple Procedures

By “simple” procedures, we mean that the following restrictions are imposed:

- Parameters are variables and expressions, not commands or procedure names.
- There are no “global” variables: All free variables of the procedure body must be formal parameters of the procedure.
- Procedures are proper, i.e., their calls are commands.
- Calls are restricted to prevent aliasing.

An additional peculiarity, which substantially simplifies reasoning about simple procedures, is that we syntactically distinguish parameters that may be modified from those that may not be.

Procedure Definitions

A *simple nonrecursive (or recursive) procedure definition* is a command of the form

$$\textbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \textbf{in}\ c'$$
$$\textbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \textbf{in}\ c',$$

where

- h is a binding occurrence of a procedure name, whose scope is c' (or c and c' in the recursive case).
- c and c' are commands.
- $v_1, \ldots, v_m; v'_1, \ldots, v'_n$ is a list of distinct variables, called *formal parameters*, that includes all of the free variables of c. The formal parameters are binding occurrences whose scope is c.
- $v_1, \ldots, v_m$ includes all of the variables modified by c.

Procedure Calls

A *procedure call* is a command of the form

$$h(w_1, \ldots, w_m; e'_1, \ldots, e'_n),$$

where

- h is a procedure name.
- $w_1, \ldots, w_m$ and $e'_1, \ldots, e'_n$ are called *actual parameters*.
- $w_1, \ldots, w_m$ are distinct variables.
- $e'_1, \ldots, e'_n$ are expressions that do not contain occurrences of the variables $w_1, \ldots, w_m$.
- The free variables of the procedure call are

  $$\mathsf{FV}(h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)) = \{w_1, \ldots, w_m\} \cup \mathsf{FV}(e'_1) \cup \cdots \cup \mathsf{FV}(e'_n)$$

  and the variables modified by the call are $w_1, \ldots, w_m$.

Hypothetical Specifications

The truth of a specification $\{p\}\ c\ \{q\}$ will depend upon an *environment*, which maps the procedure names occurring free in c into their meanings.

We define a *hypothetical specification* to have the form

$$\Gamma \vdash \{p\}\ c\ \{q\},$$

where the *context* Γ is a sequence of specifications of the form

$$\{p_0\}\ c_0\ \{q_0\}, \ldots, \{p_{n-1}\}\ c_{n-1}\ \{q_{n-1}\}.$$

We say that such a hypothetical specification is true iff $\{p\}\ c\ \{q\}$ holds for every environment in which all of the specifications in Γ hold.

Generalizing Old Inference Rules

For example,

- Strengthening Precedent (SP)

$$\frac{p \Rightarrow q \qquad \Gamma \vdash \{q\}\ c\ \{r\}}{\Gamma \vdash \{p\}\ c\ \{r\}.}$$

- Substitution (SUB)

$$\frac{\Gamma \vdash \{p\}\ c\ \{q\}}{\Gamma \vdash ((\{p\}\ c\ \{q\})/v_1 \to e_1,\ \ldots\ ,\ v_n \to e_n),}$$

  where $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

Note that substitutions do not affect procedure names.

Rules for Procedures

- Hypothesis (HYPO)

$$\frac{}{\Gamma, \{p\}\ c\ \{q\}, \Gamma' \vdash \{p\}\ c\ \{q\}.}$$

- Simple Procedures (SPROC)

$$\frac{\Gamma \vdash \{p\}\ c\ \{q\} \qquad \Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}}{\Gamma \vdash \{p'\}\ \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},}$$

where h does not occur free in any triple of Γ.

- Simple Recursive Procedures (SRPROC) (partial correctness only)

$$\frac{\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p\}\ c\ \{q\} \qquad \Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\} \vdash \{p'\}\ c'\ \{q'\}}{\Gamma \vdash \{p'\}\ \mathbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},}$$

where h does not occur free in any triple of Γ.

Some Limitations

To keep our exposition straightforward, we have ignored:

- Simultaneous recursion,
- Multiple hypotheses for the same procedure.

Two Derived Rules

From (HYPO):

- Call (CALL)

$$\frac{}{\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}, \Gamma' \vdash \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}.}$$

and from (CALL) and (SUB):

- General Call (GCALL)

$$\frac{}{\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}, \Gamma' \vdash \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},}$$

where δ is a substitution

$$\begin{aligned}\delta = v_1 \to w_1, \ldots, v_m \to w_m,\\ v'_1 \to e'_1, \ldots, v'_n \to e'_n,\\ v''_1 \to e''_1 \ldots, v''_k \to e''_k,\end{aligned}$$

which acts on all the free variables in

$$\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\},$$

and none of the variables $w_1, \ldots, w_m$ occur free in the expressions $e''_1 \ldots, e''_k$.

Annotated Specifications: Ghosts

In (GCALL):

$$\frac{}{\begin{array}{l}\Gamma, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\}, \Gamma' \vdash \\ \qquad \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},\end{array}}$$

where δ is a substitution

$$\begin{aligned}\delta = v_1 \to w_1, \ldots, v_m \to w_m, \\ v'_1 \to e'_1, \ldots, v'_n \to e'_n, \\ v''_1 \to e''_1 \ldots, v''_k \to e''_k,\end{aligned}$$

which acts on . . . . .

there may be ghost variables $v''_1, \ldots, v''_k$ that appear in δ but are not formal parameters.

We will treat $v''_1, \ldots, v''_k$ as formal ghost parameters, and $e''_1, \ldots, e''_k$ as actual ghost parameters.

For example,

$$\left.\begin{array}{l}\{\mathsf{n} \geq 0 \wedge \mathsf{r} = \mathsf{r}_0\} \\ \mathsf{multfact}(\mathsf{r};\mathsf{n}) \\ \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\}\end{array}\right\} \vdash \left\{\begin{array}{l}\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\ \mathsf{multfact}(\mathsf{r};\mathsf{n}-1) \\ \{\mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0\}\end{array}\right.$$

is an instance of (GCALL) using the substitution

$$\mathsf{r} \rightarrow \mathsf{r}, \mathsf{n} \rightarrow \mathsf{n} - 1, \mathsf{r}_0 \rightarrow \mathsf{n} \times \mathsf{r}_0.$$

The corresponding annotated specification will be

$$\left.\begin{array}{l}\{\mathsf{n} \geq 0 \wedge \mathsf{r} = \mathsf{r}_0\} \\ \mathsf{multfact}(\mathsf{r};\mathsf{n})\underline{\{\mathsf{r}_0\}} \\ \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\}\end{array}\right\} \vdash \left\{\begin{array}{l}\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\ \mathsf{multfact}(\mathsf{r};\mathsf{n}-1)\underline{\{\mathsf{n} \times \mathsf{r}_0\}} \\ \{\mathsf{r} = (\mathsf{n}-1)! \times \mathsf{n} \times \mathsf{r}_0\}.\end{array}\right.$$

Generalizing Annotation Definitions

An *annotated context* is a sequence of *annotated hypotheses*, which have the form

$$\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\},$$

where $v''_1, \ldots, v''_k$ is a list of formal ghost parameters (and all of the formal parameters, including the ghosts, are distinct).

We write $\hat{\Gamma}$ to denote an annotated context, and Γ to denote the corresponding ordinary context that is obtained by erasing the lists of ghost formal parameters. Then an annotation definition has the form:

$$\hat{\Gamma} \vdash \mathcal{A} \gg \{p\}\ c\ \{q\},$$

meaning that $\hat{\Gamma} \vdash \mathcal{A}$ is an annotated hypothetical specification proving the hypothetical specification $\Gamma \vdash \{p\}\ c\ \{q\}$.

Rules for Procedural Annotated Specifications

- General Call (GCALLan)

$$\frac{}{\begin{array}{l}\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\}, \hat{\Gamma}' \\ \qquad \vdash h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\{e''_1, \ldots, e''_k\} \\ \quad \gg \{p/\delta\}\ h(w_1, \ldots, w_m; e'_1, \ldots, e'_n)\ \{q/\delta\},\end{array}}$$

where δ is a substitution

$$\begin{array}{l}\delta = v_1 \to w_1, \ldots, v_m \to w_m, \\ \qquad v'_1 \to e'_1, \ldots, v'_n \to e'_n, \\ \qquad\quad v''_1 \to e''_1 \ldots, v''_k \to e''_k,\end{array}$$

which acts on all the free variables in

$$\{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\ \{q\},$$

and none of the variables $w_1, \ldots, w_m$ occur free in the expressions $e''_1 \ldots, e''_k$.

- Simple Procedures (SPROCan)

$$\dfrac{\begin{array}{l}\hat{\Gamma} \vdash \{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\} \\ \hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \\ \qquad\qquad \{p'\}\ \mathcal{A}'\ \{q'\} \gg \{p'\}\ c'\ \{q'\}\end{array}}{\begin{array}{l}\hat{\Gamma} \vdash \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \\ \qquad\qquad \{p\}\ \mathcal{A}\ \{q\}\ \mathbf{in}\ \{p'\}\ \mathcal{A}'\ \{q'\} \\ \quad \gg \{p'\}\ \mathbf{let}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},\end{array}}$$

where h does not occur free in any triple of $\hat{\Gamma}$.

- Simple Recursive Procedures (SRPROCan)

$$\frac{\begin{array}{r}\hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \\ \{p\}\ \mathcal{A}\ \{q\} \gg \{p\}\ c\ \{q\} \\ \hat{\Gamma}, \{p\}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\}\ \{q\} \vdash \\ \{p'\}\ \mathcal{A}'\ \{q'\} \gg \{p'\}\ c'\ \{q'\}\end{array}}{\begin{array}{r}\hat{\Gamma} \vdash \mathbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n)\{v''_1, \ldots, v''_k\} = \\ \{p\}\ \mathcal{A}\ \{q\}\ \mathbf{in}\ \{p'\}\ \mathcal{A}'\ \{q'\} \\ \gg \{p'\}\ \mathbf{letrec}\ h(v_1, \ldots, v_m; v'_1, \ldots, v'_n) = c\ \mathbf{in}\ c'\ \{q'\},\end{array}}$$

where h does not occur free in any triple of $\hat{\Gamma}$.

An Example

$$
\begin{array}{l}
\{\mathsf{z} = 10\} \\
\textbf{letrec}\ \mathsf{multfact}(\mathsf{r};\mathsf{n})\{\mathsf{r}_0\} = \\
\quad \{\mathsf{n} \geq 0 \wedge \mathsf{r} = \mathsf{r}_0\} \\
\quad \textbf{if}\ \mathsf{n} = 0\ \textbf{then} \\
\qquad \{\mathsf{n} = 0 \wedge \mathsf{r} = \mathsf{r}_0\}\ \textbf{skip}\ \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\} \\
\quad \textbf{else} \\
\qquad \{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\
\qquad\quad \left.\begin{array}{l}
\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\
\mathsf{multfact}(\mathsf{r};\mathsf{n} - 1)\{\mathsf{n} \times \mathsf{r}_0\} \\
\{\mathsf{r} = (\mathsf{n} - 1)! \times \mathsf{n} \times \mathsf{r}_0\}
\end{array}\right\} *\ \mathsf{n} - 1 \geq 0 \\
\qquad \{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = (\mathsf{n} - 1)! \times \mathsf{n} \times \mathsf{r}_0\} \\
\quad \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\} \\
\textbf{in} \\
\{5 \geq 0 \wedge \mathsf{z} = 10\} \\
\mathsf{multfact}(\mathsf{z};5)\{10\} \\
\{\mathsf{z} = 5! \times 10\}.
\end{array}
$$

How the Annotations Determine a Formal Proof

The application of (SRPROCan) to the **letrec** definition gives rise to the hypothesis

$$\{\mathsf{n} \geq 0 \wedge \mathsf{r} = \mathsf{r}_0\}\ \mathsf{multfact}(\mathsf{r}; \mathsf{n})\{\mathsf{r}_0\}\ \{\mathsf{r} = \mathsf{n}! \times \mathsf{r}_0\}.$$

By (GCALLan), the hypothesis entails

$$\begin{array}{l}\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\ \mathsf{multfact}(\mathsf{r}; \mathsf{n} - 1)\{\mathsf{n} \times \mathsf{r}_0\} \\ \{\mathsf{r} = (\mathsf{n} - 1)! \times \mathsf{n} \times \mathsf{r}_0\}.\end{array}$$

Next, since n is not modified by the call $\mathsf{multfact}(\mathsf{r}; \mathsf{n} - 1)$, the frame rule gives

$$\begin{array}{l}\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\ *\ \mathsf{n} - 1 \geq 0\} \\ \mathsf{multfact}(\mathsf{r}; \mathsf{n} - 1)\{\mathsf{n} \times \mathsf{r}_0\} \\ \{\mathsf{r} = (\mathsf{n} - 1)! \times \mathsf{n} \times \mathsf{r}_0\ *\ \mathsf{n} - 1 \geq 0\}.\end{array}$$

But the assertions here are all pure, so that the separating conjunctions can be replaced by ordinary conjunctions. Then, we can strengthen the precondition and weaken the postcondition, to obtain

$$\begin{array}{l}\{\mathsf{n} - 1 \geq 0 \wedge \mathsf{n} \times \mathsf{r} = \mathsf{n} \times \mathsf{r}_0\} \\ \mathsf{multfact}(\mathsf{r}; \mathsf{n} - 1)\{\mathsf{n} \times \mathsf{r}_0\} \\ \{\mathsf{n} - 1 \geq 0 \wedge \mathsf{r} = (\mathsf{n} - 1)! \times \mathsf{n} \times \mathsf{r}_0\}.\end{array}$$

Also, by (GCALLan), the hypothesis entails

$$\{5 \geq 0 \wedge \mathsf{z} = 10\}\ \mathsf{multfact}(\mathsf{z}; 5)\{10\}\ \{\mathsf{z} = 5! \times 10\}.$$

Some Concepts about Sequences: Images

The *image* $\{\alpha\}$ of a sequence α is the set

$$\{\, \alpha_i \mid 1 \leq i \leq \#\alpha \,\}$$

of values occurring as components of α. It satisfies the laws:

$$\begin{aligned} \{\epsilon\} &= \{\} && (1) \\ \{[x]\} &= \{x\} && (2) \\ \{\alpha{\cdot}\beta\} &= \{\alpha\} \cup \{\beta\} && (3) \\ \#\{\alpha\} &\leq \#\alpha. && (4) \end{aligned}$$

Pointwise Extension of Binary Relations

If ρ is a relation between values, then ρ^* is the relation between sets of values such that

$$S\ \rho^*\ T \text{ iff } \forall x \in S.\ \forall y \in T.\ x\ \rho\ y.$$

Pointwise extension satisfies the laws:

$$S' \subseteq S \wedge S\ \rho^*\ T \;\Rightarrow\; S'\ \rho^*\ T \tag{5}$$

$$T' \subseteq T \wedge S\ \rho^*\ T \;\Rightarrow\; S\ \rho^*\ T' \tag{6}$$

$$\{\}\ \ \rho^*\ \ T \tag{7}$$

$$S\ \ \rho^*\ \ \{\} \tag{8}$$

$$\{x\}\ \rho^*\ \{y\} \;\Leftrightarrow\; x\ \rho\ y \tag{9}$$

$$(S \cup S')\ \rho^*\ T \;\Leftrightarrow\; S\ \rho^*\ T \wedge S'\ \rho^*\ T \tag{10}$$

$$S\ \rho^*\ (T \cup T') \;\Leftrightarrow\; S\ \rho^*\ T \wedge S\ \rho^*\ T'. \tag{11}$$

The following abbreviations are also useful:

$$x\ \rho^*\ T \stackrel{\text{def}}{=} \{x\}\ \rho^*\ T \qquad S\ \rho^*\ y \stackrel{\text{def}}{=} S\ \rho^*\ \{y\}$$

Ordering

We write $\mathbf{ord}\ \alpha$ if the sequence α is ordered in nonstrict increasing order. Then $\mathbf{ord}$ satisfies

$$\#\alpha \leq 1 \;\Rightarrow\; \mathbf{ord}\ \alpha \tag{12}$$

$$\mathbf{ord}\ \alpha{\cdot}\beta \;\Leftrightarrow\; \mathbf{ord}\ \alpha \wedge \mathbf{ord}\ \beta \wedge \{\alpha\} \leq^{*} \{\beta\} \tag{13}$$

$$\mathbf{ord}\ [x]{\cdot}\alpha \;\Rightarrow\; x \leq^{*} \{[x]{\cdot}\alpha\} \tag{14}$$

$$\mathbf{ord}\ \alpha{\cdot}[x] \;\Rightarrow\; \{\alpha{\cdot}[x]\} \leq^{*} x. \tag{15}$$

Rearrangement

We say that a sequence β is a *rearrangement* of a sequence α, written $\beta \sim \alpha$, iff there is a permutation ϕ, from the domain (1 to $\#\beta$) of β to the domain (1 to $\#\alpha$) of α, such that

$$\forall k.\ 1 \leq k \leq \#\beta \text{ implies } \beta_k = \alpha_{\phi(k)}.$$

Then

$$\alpha \sim \alpha \tag{16}$$

$$\alpha \sim \beta \ \Rightarrow\ \beta \sim \alpha \tag{17}$$

$$\alpha \sim \beta \wedge \beta \sim \gamma \ \Rightarrow\ \alpha \sim \gamma \tag{18}$$

$$\alpha \sim \alpha' \wedge \beta \sim \beta' \ \Rightarrow\ \alpha{\cdot}\beta \sim \alpha'{\cdot}\beta' \tag{19}$$

$$\alpha{\cdot}\beta \sim \beta{\cdot}\alpha \tag{20}$$

$$\alpha \sim \beta \ \Rightarrow\ \{\alpha\} = \{\beta\}. \tag{21}$$

$$\alpha \sim \beta \ \Rightarrow\ \#\alpha = \#\beta. \tag{22}$$

Sorting by Merging: Lists with Explicit Lengths

The basic idea behind sorting by merging is to divide the input list segment into two roughly equal halves, sort each half recursively, and then merge the results. Unfortunately, however, one cannot divide a list segment into two halves efficiently.

A way around this difficulty is to give the lengths of the input segments to the commands for sorting and merging as explicit numbers.

We define

$$\mathsf{lseg}\ \alpha\ (e, -) \stackrel{\text{def}}{=} \exists x.\ \mathsf{lseg}\ \alpha\ (e, x).$$

Then we will define a procedure mergesort satisfying the hypothesis

$$
\begin{aligned}
H_{\mathsf{mergesort}} \stackrel{\text{def}}{=}\ & \{\mathsf{lseg}\ \alpha\,(\mathsf{i},\mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\} \\
& \mathsf{mergesort}(\mathsf{i},\mathsf{j};\mathsf{n})\{\alpha,\mathsf{j}_0\} \\
& \{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i},-) \wedge \beta \sim \alpha \wedge \mathbf{ord}\,\beta \wedge \mathsf{j} = \mathsf{j}_0\}.
\end{aligned}
$$

The subsidiary procedure merge will satisfy

$$
\begin{aligned}
H_{\mathsf{merge}} \stackrel{\text{def}}{=}\ & \{(\mathsf{lseg}\ \beta_1\,(\mathsf{i1},-) \wedge \mathbf{ord}\,\beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1}\geq 1) \\
& \quad * (\mathsf{lseg}\ \beta_2\,(\mathsf{i2},-) \wedge \mathbf{ord}\,\beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2}\geq 1)\} \\
& \mathsf{merge}(\mathsf{i};\mathsf{n1},\mathsf{n2},\mathsf{i1},\mathsf{i2})\{\beta_1,\beta_2\} \\
& \{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i},-) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,\beta\}.
\end{aligned}
$$

A Proof for mergesort

$$H_{\mathsf{mergesort}}, H_{\mathsf{merge}} \vdash \quad \{\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\}$$

$\textbf{if}\ \mathsf{n} = 1\ \textbf{then}$

$\quad \{\mathsf{lseg}\ \alpha\ (\mathsf{i}, -) \wedge \textbf{ord}\ \alpha \wedge \mathsf{i} \mapsto -, \mathsf{j}_0\}$

$\quad \mathsf{j} := [\mathsf{i} + 1]$

$\quad \{\mathsf{lseg}\ \alpha\ (\mathsf{i}, -) \wedge \textbf{ord}\ \alpha \wedge \mathsf{j} = \mathsf{j}_0\}$

$\textbf{else}$

$\quad \vdots$

$$\vdots$$

else newvar $\mathsf{n1}$ **in newvar** $\mathsf{n2}$ **in newvar** $\mathsf{i1}$ **in newvar** $\mathsf{i2}$ **in**

$$\big(\mathsf{n1} := \mathsf{n} \div 2\ ;\ \mathsf{n2} := \mathsf{n} - \mathsf{n1}\ ;\ \mathsf{i1} := \mathsf{i}\ ;$$

$$\begin{array}{l}
\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2)\ *\ \mathsf{lseg}\ \alpha_2\ (\mathsf{i}_2, \mathsf{j}_0)) \\
\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2\}
\end{array}$$

$$\left.\begin{array}{l}
\left.\begin{array}{l}
\{\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\
\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\}\ ; \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i1}, -) \wedge \beta{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta \wedge \mathsf{i2} = \mathsf{i}_2\} \\
\{\exists \beta_1.\ \mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}
\end{array}\right\} \\
\quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \wedge \#\alpha_1 = \mathsf{n1} \\
\qquad \wedge\, \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)
\end{array}\right\} \exists \alpha_1, \alpha_2, \mathsf{i}_2$$

$$\begin{array}{l}
\{\exists \alpha_1, \alpha_2, \beta_1. \\
\quad ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -)\ *\ (\mathsf{lseg}\ \alpha_2\ (\mathsf{i2}, \mathsf{j}_0)) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta_1 \\
\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2\}
\end{array}$$

$$\left.\begin{array}{l}
\left.\begin{array}{l}
\{\mathsf{lseg}\ \alpha_2\ (\mathsf{i2}, \mathsf{j}_0) \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1\} \\
\mathsf{mergesort}(\mathsf{i2}, \mathsf{j}; \mathsf{n2})\{\alpha_2, \mathsf{j}_0\}\ ; \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i2}, -) \wedge \beta{\sim}\alpha_2 \wedge \mathbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\} \\
\{\exists \beta_2.\ \mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \beta_2{\sim}\alpha_2 \wedge \mathbf{ord}\ \beta_2 \wedge \mathsf{j} = \mathsf{j}_0\}
\end{array}\right\} \\
\quad *\ (\mathsf{lseg}\ \beta_1(\mathsf{i1}, -) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \#\alpha_1 = \mathsf{n1} \\
\qquad \wedge\, \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)
\end{array}\right\} \exists \alpha_1, \alpha_2, \beta_1$$

$$\begin{array}{l}
\{\exists \alpha_1, \alpha_2, \beta_1, \beta_2. \\
\quad ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\quad *\ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \beta_2{\sim}\alpha_2 \wedge \mathbf{ord}\ \beta_2 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \\
\qquad \wedge\, \alpha = \alpha_1{\cdot}\alpha_2 \wedge \mathsf{j} = \mathsf{j}_0\} \\
\qquad \vdots
\end{array}$$

$\vdots$

$$
\begin{array}{l}
\{\exists \alpha_1, \alpha_2, \beta_1, \beta_2. \\
\;\; ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1 {\sim} \alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \# \alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\;\; * \ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \beta_2 {\sim} \alpha_2 \wedge \mathbf{ord}\ \beta_2 \wedge \# \alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \\
\quad \wedge\ \alpha = \alpha_1 {\cdot} \alpha_2 \wedge \mathsf{j} = \mathsf{j}_0\} \\
\{\exists \beta_1, \beta_2.\ ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \# \beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\;\; * \ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \# \beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)) \\
\quad \wedge\ \alpha {\sim} \beta_1 {\cdot} \beta_2 \wedge \mathsf{j} = \mathsf{j}_0\} \\
\left.\begin{array}{l}
\left.\begin{array}{l}
\{(\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \# \beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\;\; * \ (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \# \beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)\} \\
\mathsf{merge}(\mathsf{i}; \mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2})\{\beta_1, \beta_2\} \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i}, -) \wedge \beta {\sim} \beta_1 {\cdot} \beta_2 \wedge \mathbf{ord}\ \beta\}
\end{array}\right\} \\
\qquad\qquad\qquad\qquad * \ (\mathbf{emp} \wedge \alpha {\sim} \beta_1 {\cdot} \beta_2 \wedge \mathsf{j} = \mathsf{j}_0)
\end{array}\right\} \exists \beta_1, \beta_2 \\
\{\exists \beta_1, \beta_2, \beta.\ \mathsf{lseg}\ \beta (\mathsf{i}, -) \wedge \beta {\sim} \beta_1 {\cdot} \beta_2 \wedge \mathbf{ord}\ \beta \\
\;\; \wedge\ \alpha {\sim} \beta_1 {\cdot} \beta_2 \wedge \mathsf{j} = \mathsf{j}_0\} \Big) \\
\{\exists \beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i}, -) \wedge \beta {\sim} \alpha \wedge \mathbf{ord}\ \beta \wedge \mathsf{j} = \mathsf{j}_0\}.
\end{array}
$$

An Arithmetic Subtlety

In the else branch of mergesort, to determine the division of the input list segment, the variables n1 and n2 must be set to two positive integers whose sum is n.

At this point, the length n of the input list segment is at least two. Then $2 \leq n \leq 2 \times n - 2$, and since division by two is monotone:

$$1 = 2 \div 2 \leq n \div 2 \leq (2 \times n - 2) \div 2 = n - 1.$$

Thus if $n1 = n \div 2$ and $n2 = n - n1$, we have

$$1 \leq n1 \leq n - 1 \qquad 1 \leq n2 \leq n - 1 \qquad n1 + n2 = n.$$

Reasoning about the First Call of mergesort

We now expand the annotated specification of the first call of mergesort:

$$\left.\begin{array}{l}\left.\begin{array}{l}\{\mathsf{lseg}\,\alpha_1\,(\mathsf{i1},\mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\ \mathsf{mergesort}(\mathsf{i1},\mathsf{i2};\mathsf{n1})\{\alpha_1,\mathsf{i}_2\}\ ; \\ \{\exists\beta.\ \mathsf{lseg}\,\beta\,(\mathsf{i1},-) \wedge \beta{\sim}\alpha_1 \wedge \mathbf{ord}\,\beta \wedge \mathsf{i2} = \mathsf{i}_2\} \\ \{\exists\beta_1.\ \mathsf{lseg}\,\beta_1\,(\mathsf{i1},-) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\,\beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}\end{array}\right\} \\ \quad * \ (\mathsf{lseg}\,\alpha_2(\mathsf{i}_2,\mathsf{j}_0) \wedge \#\alpha_1 = \mathsf{n1} \\ \qquad \wedge\, \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\end{array}\right\} \exists\alpha_1,\alpha_2,\mathsf{i}_2$$

From the hypothesis

$$H_{\mathsf{mergesort}} \stackrel{\mathsf{def}}{=} \begin{array}{l}\{\mathsf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}_0) \wedge \#\alpha = \mathsf{n} \wedge \mathsf{n} \geq 1\} \\ \mathsf{mergesort}(\mathsf{i},\mathsf{j};\mathsf{n})\{\alpha,\mathsf{j}_0\} \\ \{\exists\beta.\ \mathsf{lseg}\,\beta\,(\mathsf{i},-) \wedge \beta \sim \alpha \wedge \mathbf{ord}\,\beta \wedge \mathsf{j} = \mathsf{j}_0\},\end{array}$$

(GCALL) is used to infer

$$\begin{array}{l}\{\mathsf{lseg}\,\alpha_1\,(\mathsf{i1},\mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\ \mathsf{mergesort}(\mathsf{i1},\mathsf{i2};\mathsf{n1})\{\alpha_1,\mathsf{i}_2\} \\ \{\exists\beta.\ \mathsf{lseg}\,\beta\,(\mathsf{i1},-) \wedge \beta \sim \alpha_1 \wedge \mathbf{ord}\,\beta \wedge \mathsf{i2} = \mathsf{i}_2\}\}.\end{array}$$

$$
\left.\begin{array}{l}
\left.\begin{array}{l}
\{\mathsf{lseg}\;\alpha_1\,(\mathsf{i1},\mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\
\mathsf{mergesort}(\mathsf{i1},\mathsf{i2};\mathsf{n1})\{\alpha_1,\mathsf{i}_2\}\;; \\
\{\exists\beta.\;\mathsf{lseg}\;\beta\,(\mathsf{i1},-) \wedge \beta{\sim}\alpha_1 \wedge \mathbf{ord}\,\beta \wedge \mathsf{i2} = \mathsf{i}_2\} \\
\{\exists\beta_1.\;\mathsf{lseg}\;\beta_1\,(\mathsf{i1},-) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\,\beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}
\end{array}\right\} \\
\quad *\;\;(\mathsf{lseg}\;\alpha_2(\mathsf{i}_2,\mathsf{j}_0) \wedge \#\alpha_1 = \mathsf{n1} \\
\qquad \wedge\, \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)
\end{array}\right\}\exists\alpha_1,\alpha_2,\mathsf{i}_2
$$

Then β is renamed β_1 in the postcondition:

$$
\begin{array}{l}
\{\mathsf{lseg}\;\alpha_1\,(\mathsf{i1},\mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\
\mathsf{mergesort}(\mathsf{i1},\mathsf{i2};\mathsf{n1})\{\alpha_1,\mathsf{i}_2\} \\
\{\exists\beta_1.\;\mathsf{lseg}\;\beta_1\,(\mathsf{i1},-) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\,\beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}.
\end{array}
$$

$$\left.\begin{array}{l}\left.\begin{array}{l}\{\mathsf{lseg}\ \alpha_1\,(\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\ \mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\}\ ; \\ \{\exists\beta.\ \mathsf{lseg}\ \beta\,(\mathsf{i1}, -) \wedge \beta{\sim}\alpha_1 \wedge \mathbf{ord}\,\beta \wedge \mathsf{i2} = \mathsf{i}_2\} \\ \{\exists\beta_1.\ \mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\,\beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}\end{array}\right\} \\ \quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \wedge \#\alpha_1 = \mathsf{n1} \\ \qquad \wedge\, \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\end{array}\right\} \exists\alpha_1, \alpha_2, \mathsf{i}_2$$

Next, the frame rule is used to infer

$$\begin{array}{l}\{(\mathsf{lseg}\ \alpha_1\,(\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\ \quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\ \quad \wedge\, \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\} \\ \mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\} \\ \{(\exists\beta_1.\ \mathsf{lseg}\ \beta_1\,(\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\,\beta_1 \wedge \mathsf{i2} = \mathsf{i}_2) \\ \quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\ \quad \wedge\, \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\}.\end{array}$$

$$\left.\left.\begin{array}{l}\{\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1\} \\ \mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\}\ ; \\ \{\exists\beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i1}, -) \wedge \beta{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta \wedge \mathsf{i2} = \mathsf{i}_2\} \\ \{\exists\beta_1.\ \mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2\}\end{array}\right\} \begin{array}{l} \\ \ast\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \wedge \#\alpha_1 = \mathsf{n1} \\ \quad \wedge\ \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\end{array}\right\} \exists\alpha_1, \alpha_2, \mathsf{i}_2$$

Then the rule (EQ) for existential quantification gives

$$\begin{array}{l}\{\exists\alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\ \quad \ast\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\ \quad \wedge\ \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2))\} \\ \mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\} \\ \{\exists\alpha_1, \alpha_2, \mathsf{i}_2.\ (\exists\beta_1.\ \mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1{\sim}\alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2) \\ \quad \ast\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\ \quad \wedge\ \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2))\}.\end{array}$$

Finally, $\mathsf{i2} = \mathsf{i}_2$ is used to eliminate i_2 in the postcondition, and pure terms are rearranged in both the pre- and postconditions:

$$
\begin{array}{l}
\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2)\ *\ \mathsf{lseg}\ \alpha_2\ (\mathsf{i}_2, \mathsf{j}_0)) \\
\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2\} \\
\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\mathsf{lseg}\ \alpha_1\ (\mathsf{i1}, \mathsf{i}_2) \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1) \\
\quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\
\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\} \\
\mathsf{mergesort}(\mathsf{i1}, \mathsf{i2}; \mathsf{n1})\{\alpha_1, \mathsf{i}_2\} \\
\{\exists \alpha_1, \alpha_2, \mathsf{i}_2.\ (\exists \beta_1.\ \mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \wedge \mathsf{i2} = \mathsf{i}_2) \\
\quad *\ (\mathsf{lseg}\ \alpha_2(\mathsf{i}_2, \mathsf{j}_0) \\
\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2)\} \\
\{\exists \alpha_1, \alpha_2, \beta_1. \\
\quad ((\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -)\ *\ (\mathsf{lseg}\ \alpha_2\ (\mathsf{i2}, \mathsf{j}_0)) \wedge \beta_1 \sim \alpha_1 \wedge \mathbf{ord}\ \beta_1 \\
\quad \wedge \#\alpha_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1 \wedge \#\alpha_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1 \wedge \alpha = \alpha_1{\cdot}\alpha_2\}.
\end{array}
$$

merge with goto's

$\mathsf{merge}(\mathsf{i}; \mathsf{n1}, \mathsf{n2}, \mathsf{i1}, \mathsf{i2})\{\beta_1, \beta_2\} =$

$\quad$ **newvar** a1 **in newvar** a2 **in newvar** j **in**

$\quad\quad$ (a1 := [i1] ; a2 := [i2] ;

$\quad\quad$ **if** a1 $\leq$ a2 **then** i := i1 ; **goto** $\ell 1$ **else** i := i2 ; **goto** $\ell 2$;

$\ell 1$: **if** n1 = 1 **then** [i1 + 1] := i2 ; **goto** out **else**

$\quad\quad$ n1 := n1 − 1 ; j := i1 ; i1 := [j + 1] ; a1 := [i1];

$\quad\quad$ **if** a1 $\leq$ a2 **then goto** $\ell 1$ **else** [j + 1] := i2 ; **goto** $\ell 2$;

$\ell 2$: **if** n2 = 1 **then** [i2 + 1] := i1 ; **goto** out **else**

$\quad\quad$ n2 := n2 − 1 ; j := i2 ; i2 := [j + 1] ; a2 := [i2];

$\quad\quad$ **if** a2 $\leq$ a1 **then goto** $\ell 2$ **else** [j + 1] := i1 ; **goto** $\ell 1$;

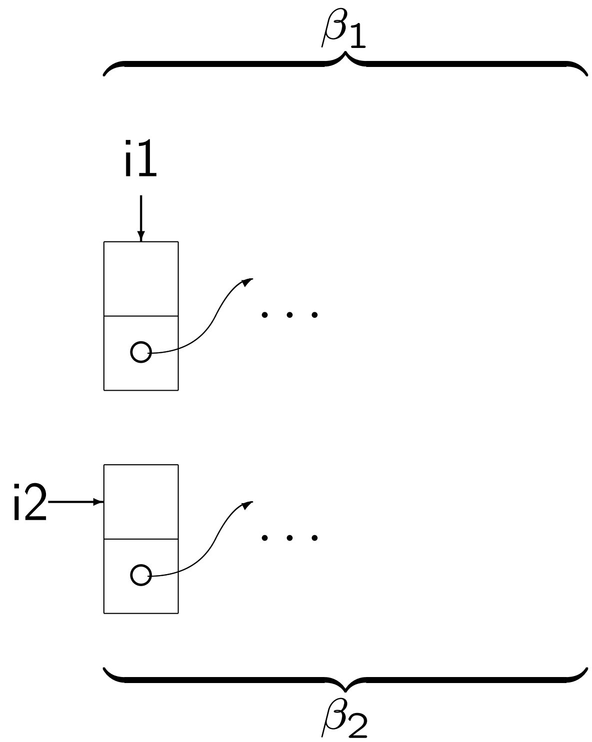
out:)

A Proof for merge with goto's

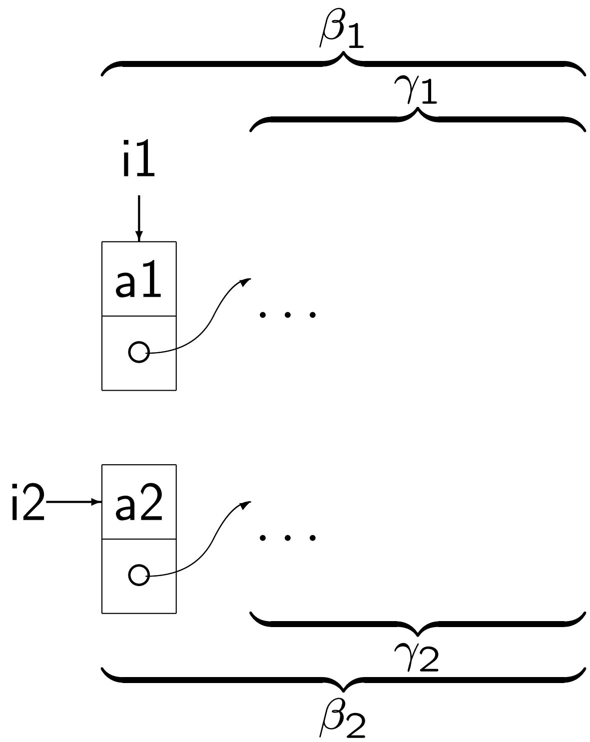
$$
\begin{array}{l}
\mathsf{merge}(\mathsf{i};\mathsf{n1},\mathsf{n2},\mathsf{i1},\mathsf{i2})\{\beta_1,\beta_2\} = \\
\quad \{(\mathsf{lseg}\,\beta_1\,(\mathsf{i1},-) \wedge \mathbf{ord}\,\beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1}{\geq}1) \\
\qquad * (\mathsf{lseg}\,\beta_2\,(\mathsf{i2},-) \wedge \mathbf{ord}\,\beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2}{\geq}1)\} \\
\quad \mathbf{newvar}\ \mathsf{a1}\ \mathbf{in}\ \mathbf{newvar}\ \mathsf{a2}\ \mathbf{in}\ \mathbf{newvar}\ \mathsf{j}\ \mathbf{in} \\
\qquad \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ; \\
\qquad \mathbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \mathbf{then}\ \mathsf{i} := \mathsf{i1}\ ;\ \mathbf{goto}\ \ell 1\ \mathbf{else}\ \mathsf{i} := \mathsf{i2}\ ;\ \mathbf{goto}\ \ell 2\ ; \\
\ell 1\!: \{\exists\beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
\qquad\quad (\mathsf{lseg}\,\beta\,(\mathsf{i},\mathsf{i1})\ *\ \mathsf{i1} \mapsto \mathsf{a1},\mathsf{j1}\ *\ \mathsf{lseg}\,\gamma_1\,(\mathsf{j1},-) \\
\qquad\qquad *\ \mathsf{i2} \mapsto \mathsf{a2},\mathsf{j2}\ *\ \mathsf{lseg}\,\gamma_2\,(\mathsf{j2},-)) \\
\qquad\quad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \mathsf{a1} \leq \mathsf{a2} \\
\qquad\quad \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \\
\qquad\quad \wedge \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\} \\
\qquad \mathbf{if}\ \mathsf{n1} = 1\ \mathbf{then}\ [\mathsf{i1}+1] := \mathsf{i2}\ ;\ \mathbf{goto}\ \mathsf{out}\ \mathbf{else} \\
\qquad \mathsf{n1} := \mathsf{n1} - 1\ ;\ \mathsf{j} := \mathsf{i1}\ ;\ \mathsf{i1} := [\mathsf{j}+1]\ ;\ \mathsf{a1} := [\mathsf{i1}]; \\
\qquad \{\exists\beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2. \\
\qquad\quad (\mathsf{lseg}\,\beta\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a1}',\mathsf{i1}\ *\ \mathsf{i1} \mapsto \mathsf{a1},\mathsf{j1}\ *\ \mathsf{lseg}\,\gamma_1'\,(\mathsf{j1},-) \\
\qquad\qquad *\ \mathsf{i2} \mapsto \mathsf{a2},\mathsf{j2}\ *\ \mathsf{lseg}\,\gamma_2\,(\mathsf{j2},-)) \\
\qquad\quad \wedge \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \\
\qquad\quad \wedge \beta{\cdot}\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,(\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \\
\qquad\quad \wedge \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\} \\
\qquad \mathbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \mathbf{then}\ \mathbf{goto}\ \ell 1\ \mathbf{else}\ [\mathsf{j}+1] := \mathsf{i2}\ ;\ \mathbf{goto}\ \ell 2\ ; \\
\qquad\quad \vdots
\end{array}
$$

$$\vdots$$

$$
\begin{array}{l}
\ell 2\text{: } \{\exists\beta, \mathsf{a2}, \mathsf{j2}, \gamma_2, \mathsf{j1}, \gamma_1. \\
\quad (\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{i2})\ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -) \\
\quad\quad *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -)) \\
\quad \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \mathsf{a2} \leq \mathsf{a1} \\
\quad \wedge \beta{\cdot}\mathsf{a2}{\cdot}\gamma_2{\cdot}\mathsf{a1}{\cdot}\gamma_1 \sim \beta_2{\cdot}\beta_1 \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \\
\quad \wedge \mathbf{ord}\ \beta \wedge \{\beta\} \leq^* \{\mathsf{a2}{\cdot}\gamma_2\} \cup \{\mathsf{a1}{\cdot}\gamma_1\}\} \\
\mathbf{if}\ \mathsf{n2} = 1\ \mathbf{then}\ [\mathsf{i2} + 1] := \mathsf{i1}\ ;\ \mathbf{goto}\ \mathsf{out}\ \mathbf{else} \\
\mathsf{n2} := \mathsf{n2} - 1\ ;\ \mathsf{j} := \mathsf{i2}\ ;\ \mathsf{i2} := [\mathsf{j} + 1]\ ;\ \mathsf{a2} := [\mathsf{i2}]; \\
\{\exists\beta, \mathsf{a2}', \mathsf{j2}, \gamma_2', \mathsf{j1}, \gamma_1. \\
\quad (\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a2}', \mathsf{i2}\ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2'\ (\mathsf{j2}, -) \\
\quad\quad *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -)) \\
\quad \wedge \#\gamma_2' = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1 \\
\quad \wedge \beta{\cdot}\mathsf{a2}'{\cdot}\mathsf{a2}{\cdot}\gamma_2'{\cdot}\mathsf{a1}{\cdot}\gamma_1 \sim \beta_2{\cdot}\beta_1 \wedge \mathbf{ord}\,(\mathsf{a2}'{\cdot}\mathsf{a2}{\cdot}\gamma_2') \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \\
\quad \wedge \mathbf{ord}\ \beta \wedge \{\beta\} \leq^* \{\mathsf{a2}'{\cdot}\mathsf{a2}{\cdot}\gamma_2'\} \cup \{\mathsf{a1}{\cdot}\gamma_1\}\} \\
\mathbf{if}\ \mathsf{a2} \leq \mathsf{a1}\ \mathbf{then\ goto}\ \ell 2\ \mathbf{else}\ [\mathsf{j} + 1] := \mathsf{i1}\ ;\ \mathbf{goto}\ \ell 1\ ; \\
\mathsf{out}\text{: } \Big) \\
\{\exists\beta.\ \mathsf{lseg}\ \beta\ (\mathsf{i}, -) \wedge \beta \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\ \beta\}.
\end{array}
$$



$$\{(\mathsf{lseg}\ \beta_1\ (\mathsf{i1}, -) \wedge \mathbf{ord}\ \beta_1 \wedge \#\beta_1 = \mathsf{n1} \wedge \mathsf{n1} \geq 1)$$
$$* (\mathsf{lseg}\ \beta_2\ (\mathsf{i2}, -) \wedge \mathbf{ord}\ \beta_2 \wedge \#\beta_2 = \mathsf{n2} \wedge \mathsf{n2} \geq 1)\}$$



newvar a1 **in newvar** a2 **in newvar** j **in**

$\big($a1 := [i1] ; a2 := [i2] ;

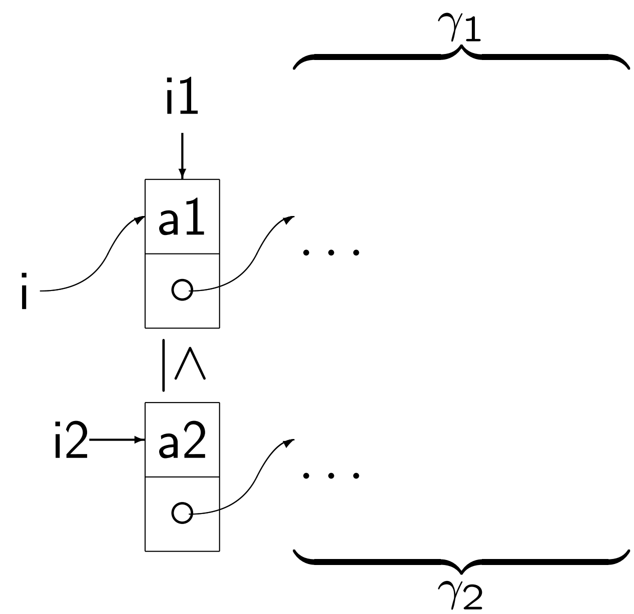
$\{\exists \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2.$

$(\mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{lseg}\, \gamma_1\, (\mathsf{j1}, -)$

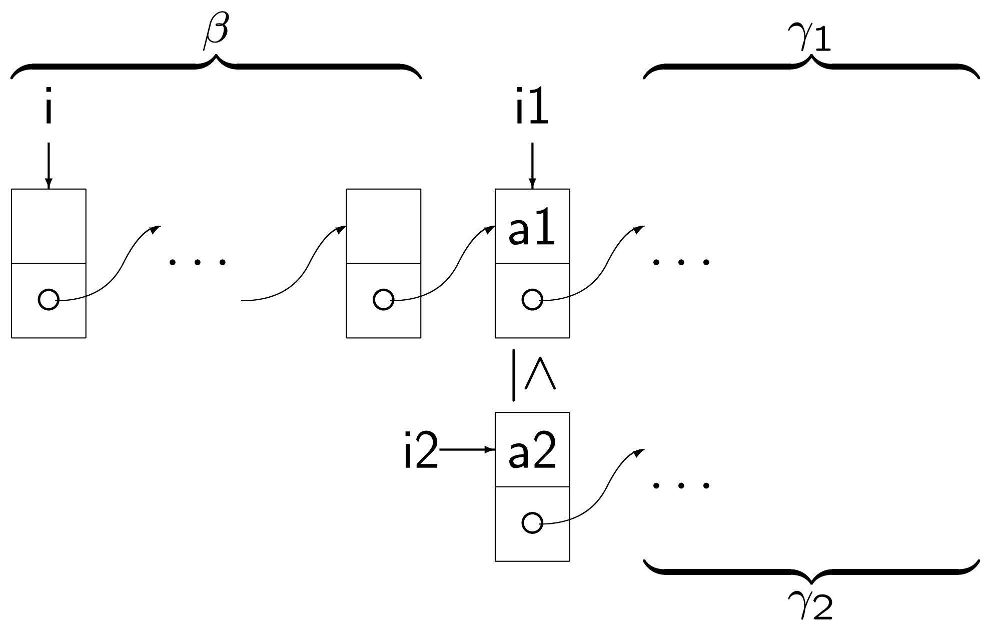
$* \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\, \gamma_2\, (\mathsf{j2}, -))$

$\wedge \ \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1$

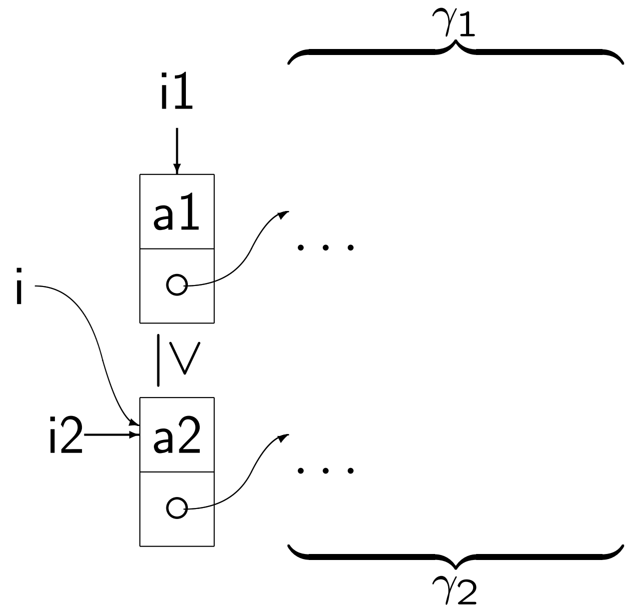
$\wedge \ \mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 = \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2)\}$



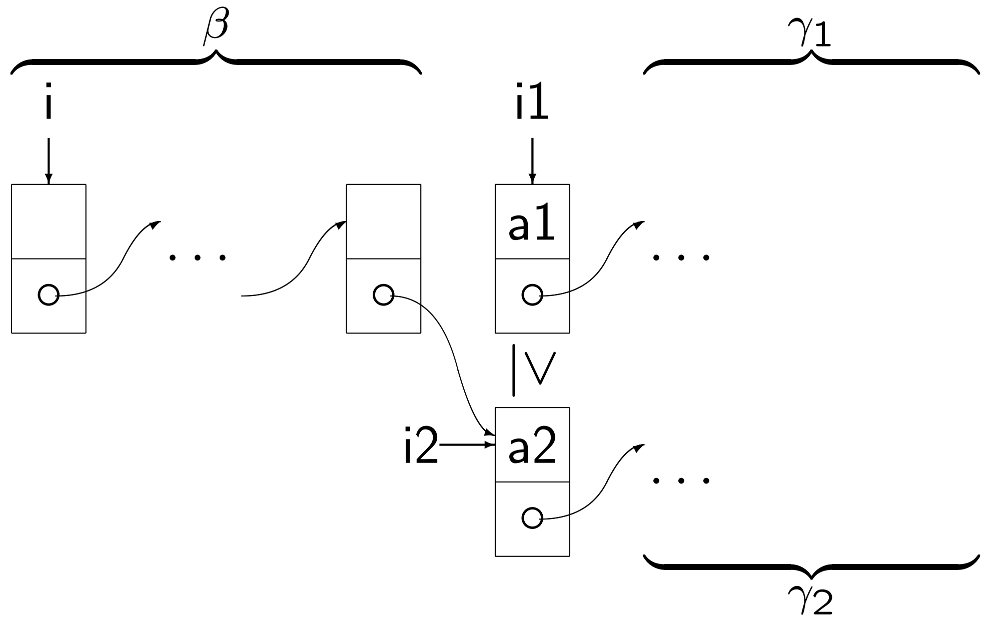
$$
\begin{array}{l}
\textbf{newvar}\ \mathsf{a1}\ \textbf{in}\ \textbf{newvar}\ \mathsf{a2}\ \textbf{in}\ \textbf{newvar}\ \mathsf{j}\ \textbf{in} \\
\quad \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ; \\
\quad \textbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \textbf{then}\ \mathsf{i} := \mathsf{i1}; \\
\quad \{\exists \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
\qquad ((\mathsf{i} = \mathsf{i1} \wedge \textbf{emp})\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -) \\
\qquad\quad *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -)) \\
\qquad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \mathsf{a1} \leq \mathsf{a2} \\
\qquad \wedge \mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 = \beta_1{\cdot}\beta_2 \wedge \textbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \textbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2)\} \\
\quad \textbf{goto}\ \ell 1
\end{array}
$$



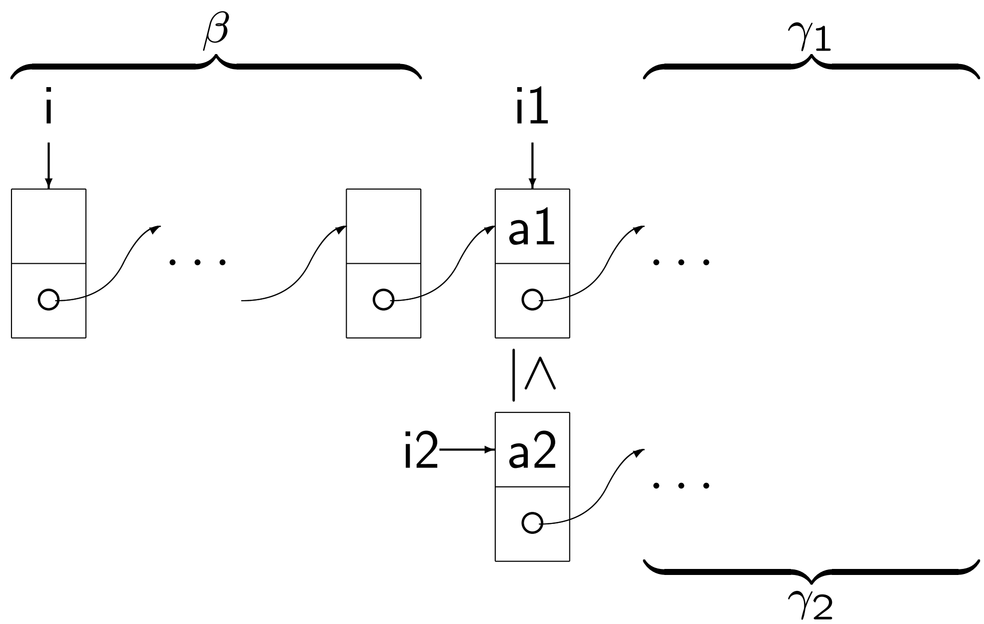
$$
\begin{aligned}
&\ell 1\text{: } \{\exists \beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
&\qquad (\mathsf{lseg}\,\beta\,(\mathsf{i}, \mathsf{i1}) \;*\; \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \;*\; \mathsf{lseg}\,\gamma_1\,(\mathsf{j1}, -) \\
&\qquad\quad *\; \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \;*\; \mathsf{lseg}\,\gamma_2\,(\mathsf{j2}, -)) \\
&\qquad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \mathsf{a1} \leq \mathsf{a2} \\
&\qquad \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \\
&\qquad \wedge \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}
\end{aligned}
$$



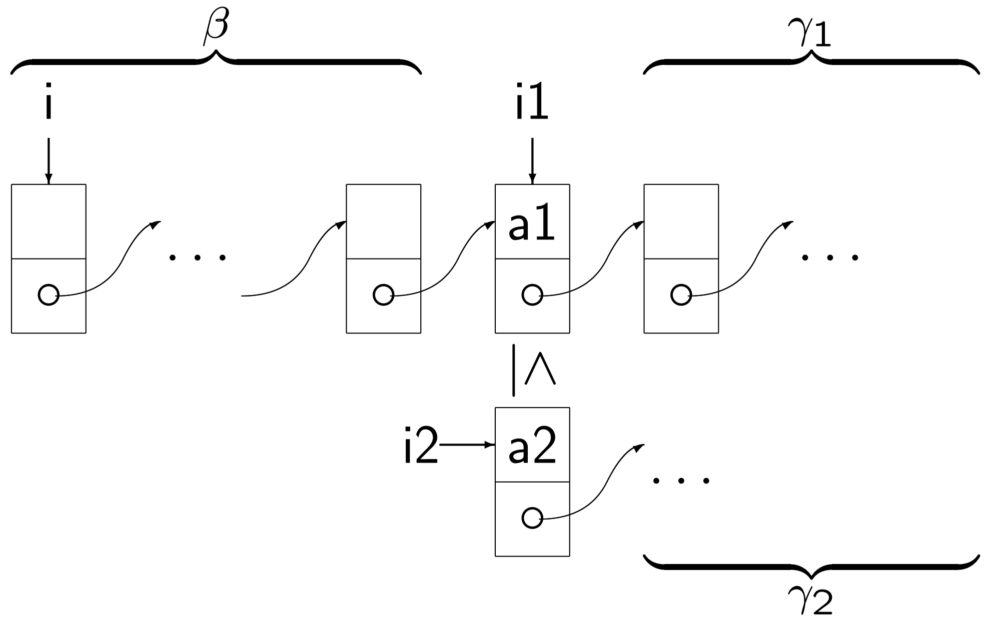
$$
\begin{array}{l}
\textbf{newvar}\ \mathsf{a1}\ \textbf{in}\ \textbf{newvar}\ \mathsf{a2}\ \textbf{in}\ \textbf{newvar}\ \mathsf{j}\ \textbf{in} \\
\quad \big(\mathsf{a1} := [\mathsf{i1}]\ ;\ \mathsf{a2} := [\mathsf{i2}]\ ; \\
\quad \textbf{if}\ \mathsf{a1} \leq \mathsf{a2}\ \textbf{then}\ \mathsf{i} := \mathsf{i1}\ ;\ \textbf{goto}\ \ell 1\ \textbf{else}\ \mathsf{i} := \mathsf{i2}; \\
\quad \{\exists \mathsf{a2}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
\qquad ((\mathsf{i} = \mathsf{i1} \wedge \mathbf{emp}) \ast \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ast \mathsf{lseg}\ \gamma_1\,(\mathsf{j1}, -) \\
\qquad\quad \ast\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ast \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)) \\
\qquad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \mathsf{a2} \leq \mathsf{a1} \\
\qquad \wedge\ \mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 = \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2)\} \\
\quad \textbf{goto}\ \ell 2\ ;
\end{array}
$$



$$
\begin{aligned}
\ell 2\colon \{&\exists \beta, \mathsf{a2}, \mathsf{j2}, \gamma_2, \mathsf{j1}, \gamma_1 .\\
&(\mathsf{lseg}\,\beta\,(\mathsf{i}, \mathsf{i2}) \ * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\,\gamma_2\,(\mathsf{j2}, -)\\
&\quad * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{lseg}\,\gamma_1\,(\mathsf{j1}, -))\\
&\wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \mathsf{a2} \leq \mathsf{a1}\\
&\wedge \beta{\cdot}\mathsf{a2}{\cdot}\gamma_2{\cdot}\mathsf{a1}{\cdot}\gamma_1 {\sim} \beta_2{\cdot}\beta_1 \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1)\\
&\wedge \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a2}{\cdot}\gamma_2\} \cup \{\mathsf{a1}{\cdot}\gamma_1\}\}
\end{aligned}
$$

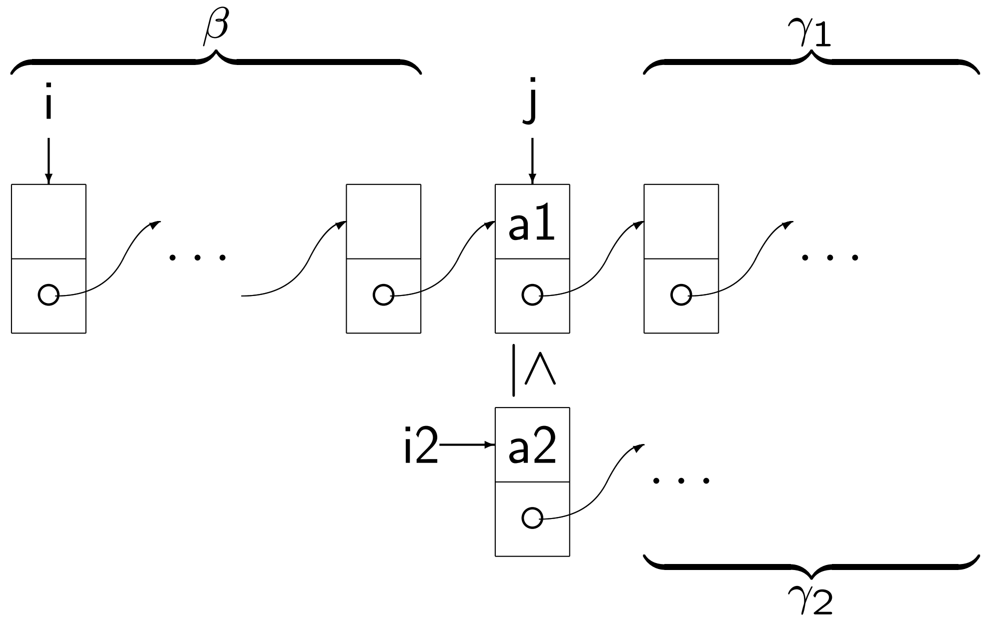


$$
\begin{aligned}
&\ell 1\text{: } \{\exists \beta, \mathsf{a1}, \mathsf{j1}, \gamma_1, \mathsf{j2}, \gamma_2. \\
&\qquad (\mathsf{lseg}\ \beta\ (\mathsf{i}, \mathsf{i1})\ *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1}\ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -) \\
&\qquad\quad *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2}\ *\ \mathsf{lseg}\ \gamma_2\ (\mathsf{j2}, -)) \\
&\qquad \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \mathsf{a1} \leq \mathsf{a2} \\
&\qquad \wedge \beta{\cdot}\mathsf{a1}{\cdot}\gamma_1{\cdot}\mathsf{a2}{\cdot}\gamma_2 \sim \beta_1{\cdot}\beta_2 \wedge \mathbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1) \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \\
&\qquad \wedge \mathbf{ord}\ \beta \wedge \{\beta\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}
\end{aligned}
$$



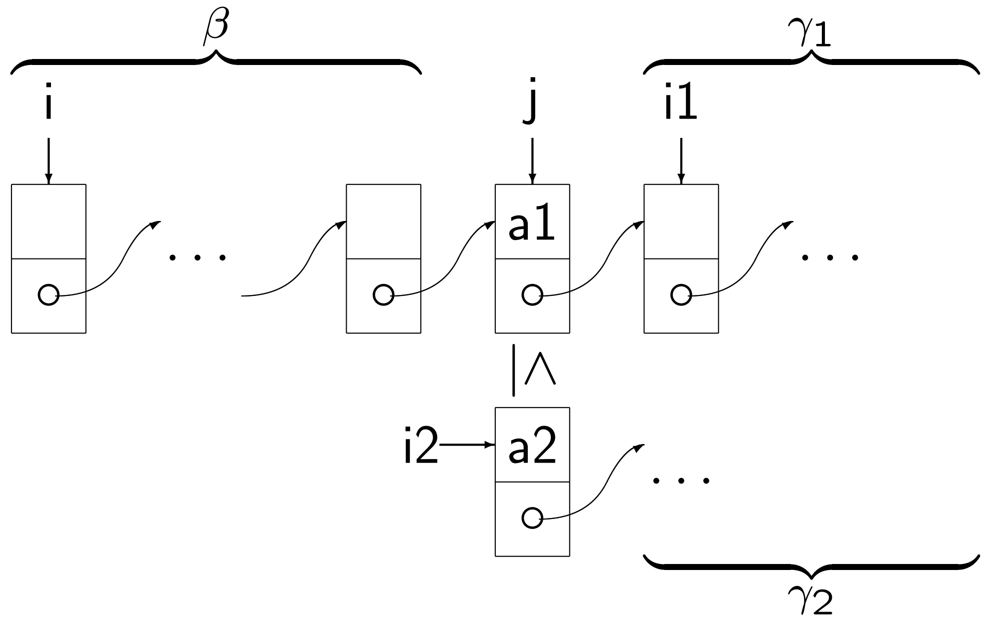
if n1 $=$ 1 **then** [i1 $+$ 1] $:=$ i2 ; **goto** out **else**

n1 $:=$ n1 $-$ 1;

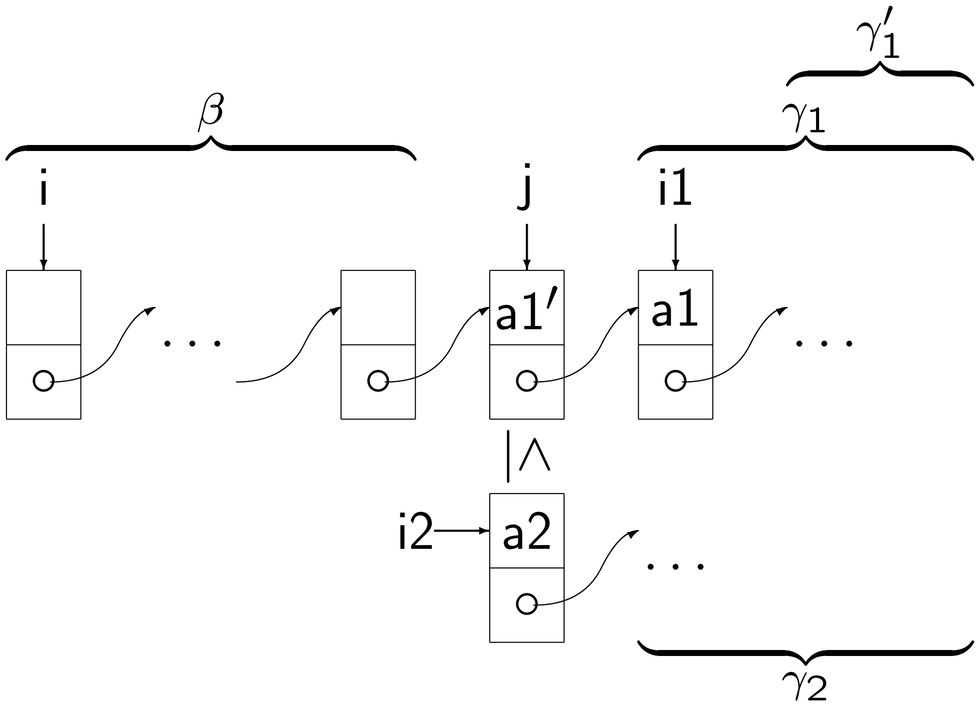


$\mathbf{if}\ \mathsf{n1} = 1\ \mathbf{then}\ [\mathsf{i1}+1] := \mathsf{i2}\ ;\ \mathbf{goto}\ \mathsf{out}\ \mathbf{else}$

$\mathsf{n1} := \mathsf{n1} - 1\ ;\ \mathsf{j} := \mathsf{i1};$



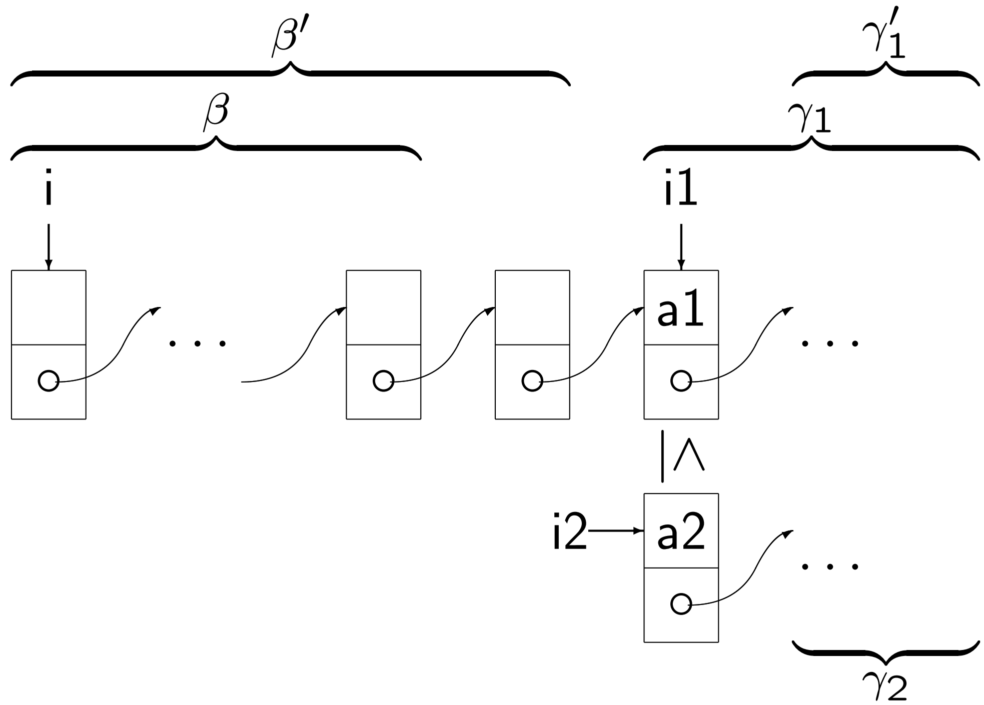
if n1 = 1 **then** [i1 + 1] := i2 ; **goto** out **else**
n1 := n1 − 1 ; j := i1 ; i1 := [j + 1];



$$\textbf{if } \mathsf{n1} = 1 \textbf{ then } [\mathsf{i1}+1] := \mathsf{i2} \,;\, \textbf{goto } \mathsf{out} \textbf{ else}$$

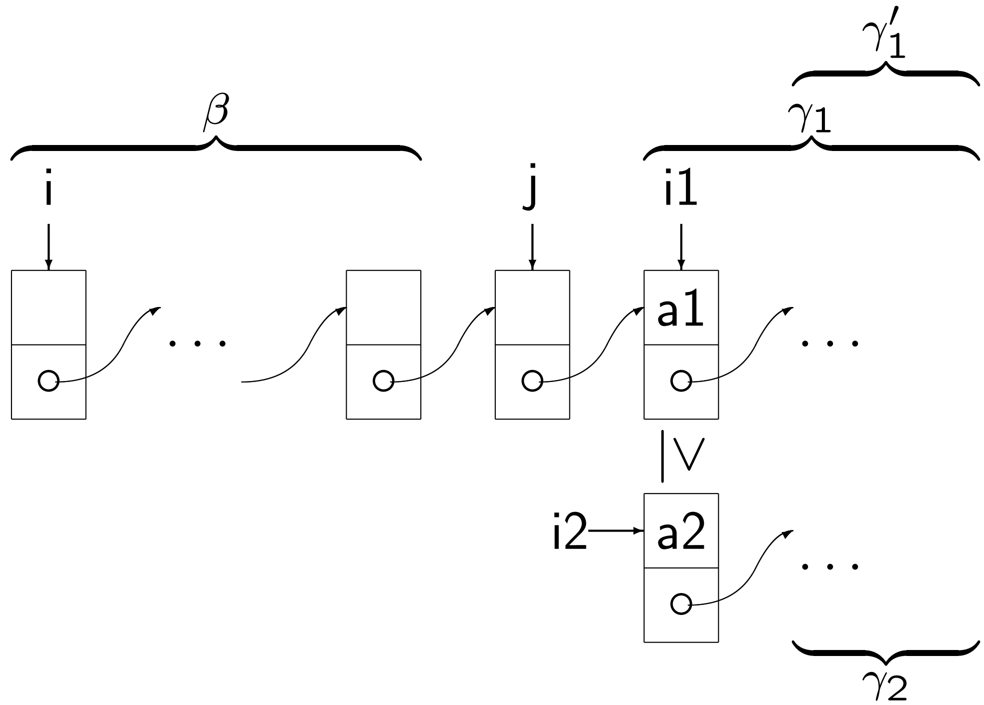
$$\mathsf{n1} := \mathsf{n1} - 1 \,;\, \mathsf{j} := \mathsf{i1} \,;\, \mathsf{i1} := [\mathsf{j}+1] \,;\, \mathsf{a1} := [\mathsf{i1}];$$

$$\begin{aligned}
&\{\exists \beta, \mathsf{a1}', \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2. \\
&\quad (\mathsf{lseg}\ \beta\,(\mathsf{i},\mathsf{j}) \ * \ \mathsf{j} \mapsto \mathsf{a1}', \mathsf{i1} \ * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{lseg}\ \gamma_1'\,(\mathsf{j1}, -) \\
&\qquad * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)) \\
&\quad \wedge \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \\
&\quad \wedge \beta{\cdot}\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \\
&\quad \left.\begin{aligned}&\wedge \mathbf{ord}\,(\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \\ &\wedge \mathbf{ord}\ \beta \wedge \{\beta\} \leq^* \{\mathsf{a1}'{\cdot}\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}\end{aligned}\right\} (A)
\end{aligned}$$

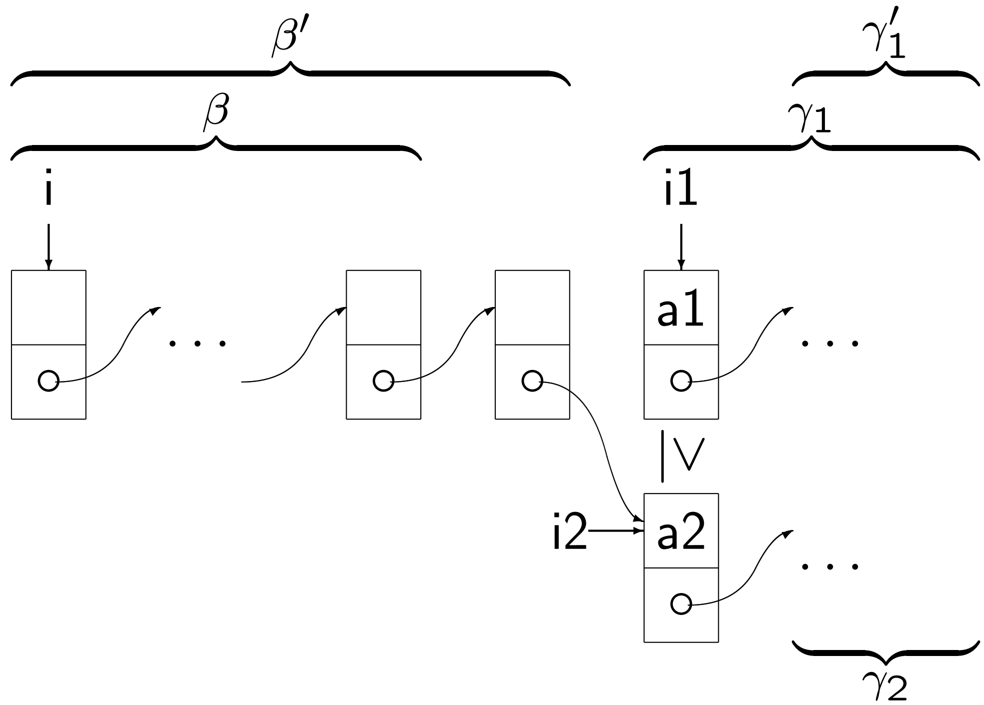


$$\textbf{if } \mathsf{a1} \leq \mathsf{a2} \textbf{ then goto } \ell 1$$

$$\begin{aligned}
\ell 1\colon \{&\exists \beta', \mathsf{a1}, \mathsf{j1}, \gamma_1', \mathsf{j2}, \gamma_2. \\
&(\mathsf{lseg}\ \beta'\,(\mathsf{i}, \mathsf{i1}) \ * \ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ * \ \mathsf{lseg}\ \gamma_1'\,(\mathsf{j1}, -) \\
&\quad * \ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ * \ \mathsf{lseg}\ \gamma_2\,(\mathsf{j2}, -)) \\
&\wedge \#\gamma_1' = \mathsf{n1} - 1 \wedge \#\gamma_2 = \mathsf{n2} - 1 \wedge \mathsf{a1} \leq \mathsf{a2} \\
&\wedge \beta'{\cdot}\mathsf{a1}{\cdot}\gamma_1'{\cdot}\mathsf{a2}{\cdot}\gamma_2 {\sim} \beta_1{\cdot}\beta_2 \\
&\left.\begin{aligned}
&\wedge \textbf{ord}\,(\mathsf{a1}{\cdot}\gamma_1') \wedge \textbf{ord}\,(\mathsf{a2}{\cdot}\gamma_2) \\
&\wedge \textbf{ord}\,\beta' \wedge \{\beta'\} \leq^* \{\mathsf{a1}{\cdot}\gamma_1'\} \cup \{\mathsf{a2}{\cdot}\gamma_2\}\}
\end{aligned}\right\}(B)
\end{aligned}$$



if $\mathsf{a1} \leq \mathsf{a2}$ **then goto** $\ell 1$ **else**



$$\textbf{if } \mathsf{a1} \leq \mathsf{a2} \textbf{ then goto } \ell 1 \textbf{ else } [\mathsf{j}+1] := \mathsf{i2} \,;\, \textbf{goto } \ell 2 \,;$$

$$\begin{aligned}
\ell 2\text{: } \{\exists \beta', \mathsf{a2}, \mathsf{j2}, \gamma_2', \mathsf{j1}, \gamma_1.\ & \\
& (\mathsf{lseg}\ \beta'\ (\mathsf{i}, \mathsf{i2}) \ *\ \mathsf{i2} \mapsto \mathsf{a2}, \mathsf{j2} \ *\ \mathsf{lseg}\ \gamma_2'\ (\mathsf{j2}, -) \\
& \quad *\ \mathsf{i1} \mapsto \mathsf{a1}, \mathsf{j1} \ *\ \mathsf{lseg}\ \gamma_1\ (\mathsf{j1}, -)) \\
& \wedge \#\gamma_2' = \mathsf{n2} - 1 \wedge \#\gamma_1 = \mathsf{n1} - 1 \wedge \mathsf{a2} \leq \mathsf{a1} \\
& \wedge \beta' \cdot \mathsf{a2} \cdot \gamma_2' \cdot \mathsf{a1} \cdot \gamma_1 \sim \beta_2 \cdot \beta_1 \wedge \mathbf{ord}\ (\mathsf{a2} \cdot \gamma_2') \wedge \mathbf{ord}\ (\mathsf{a1} \cdot \gamma_1) \\
& \wedge \mathbf{ord}\ \beta' \wedge \{\beta'\} \leq^* \{\mathsf{a2} \cdot \gamma_2'\} \cup \{\mathsf{a1} \cdot \gamma_1\}\}
\end{aligned}$$

The Ordering Argument

$$\left.\begin{array}{l}\mathbf{ord}\,(\mathsf{a1}'\cdot\mathsf{a1}\cdot\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}\cdot\gamma_2) \\ \quad \wedge\, \mathbf{ord}\,\beta \wedge \{\beta\} \leq^* \{\mathsf{a1}'\cdot\mathsf{a1}\cdot\gamma_1'\} \cup \{\mathsf{a2}\cdot\gamma_2\}\}\end{array}\right\}(A)$$

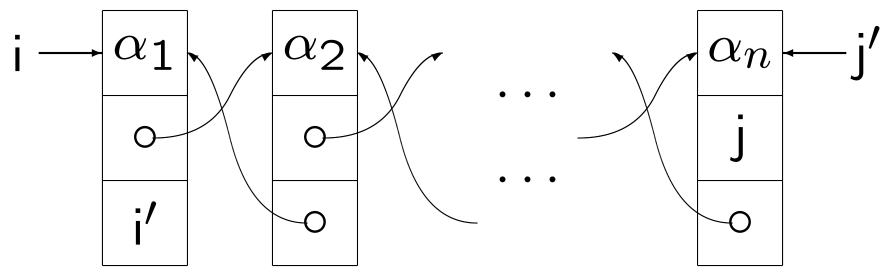
$$\wedge\, \mathsf{a1} \leq \mathsf{a2} \Rightarrow$$

$$\left.\begin{array}{l}\mathbf{ord}\,(\mathsf{a1}\cdot\gamma_1') \wedge \mathbf{ord}\,(\mathsf{a2}\cdot\gamma_2) \\ \quad \wedge\, \mathbf{ord}\,\beta\cdot\mathsf{a1}' \wedge \{\beta\cdot\mathsf{a1}'\} \leq^* \{\mathsf{a1}\cdot\gamma_1'\} \cup \{\mathsf{a2}\cdot\gamma_2\}\}\end{array}\right\}(B)$$

1.	$\mathbf{ord}(\mathsf{a1}'\cdot\mathsf{a1}\cdot\gamma_1')$	(assumption)
∗2.	$\mathbf{ord}(\mathsf{a1}\cdot\gamma_1')$	(13),1
3.	$\mathsf{a1}' \leq^* \{\mathsf{a1}\cdot\gamma_1'\}$	(13),1
4.	$\mathsf{a1}' \leq \mathsf{a1}$	(6),3
5.	$\mathsf{a1} \leq \mathsf{a2}$	(assumption)
6.	$\mathsf{a1}' \leq \mathsf{a2}$	(transitivity),4,5
∗7.	$\mathbf{ord}(\mathsf{a2}\cdot\gamma_2)$	(assumption)
8.	$\mathsf{a1}' \leq^* \{\mathsf{a2}\cdot\gamma_2\}$	(14),6,7
9.	$\mathsf{a1}' \leq^* \{\mathsf{a1}\cdot\gamma_1'\} \cup \{\mathsf{a2}\cdot\gamma_2\}$	(11),3,8
10.	$\{\beta\} \leq^* \{\mathsf{a1}'\cdot\mathsf{a1}\cdot\gamma_1'\} \cup \{\mathsf{a2}\cdot\gamma_2\}$	(assumption)
11.	$\{\beta\} \leq^* \{\mathsf{a1}\cdot\gamma_1'\} \cup \{\mathsf{a2}\cdot\gamma_2\}$	(3),(6),10
∗12.	$\{\beta\cdot\mathsf{a1}'\} \leq^* \{\mathsf{a1}\cdot\gamma_1'\} \cup \{\mathsf{a2}\cdot\gamma_2\}$	(10),(3),9,11
13.	$\mathbf{ord}\,\beta$	(assumption)
14.	$\mathbf{ord}\,\mathsf{a1}'$	(12)
15.	$\{\beta\} \leq^* \mathsf{a1}'$	(3),(6),10
∗16.	$\mathbf{ord}(\beta\cdot\mathsf{a1}')$	(13),13,14,15

Doubly-Linked List Segments

dlseg α (i, i′, j, j′):



is defined by

$$\mathsf{dlseg}\ \epsilon\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j} \wedge \mathsf{i}' = \mathsf{j}'$$

$$\mathsf{dlseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}, \mathsf{i}' \ * \ \mathsf{dlseg}\ \alpha\ (\mathsf{j}, \mathsf{i}, \mathsf{k}, \mathsf{k}'),$$

Properties

$$\begin{aligned}
\mathsf{dlseg\ a}\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') &\Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j},\mathsf{i}' \wedge \mathsf{i} = \mathsf{j}' \\
\mathsf{dlseg}\ \alpha{\cdot}\beta\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\Leftrightarrow \exists \mathsf{j},\mathsf{j}'.\ \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') * \mathsf{dlseg}\ \beta\,(\mathsf{j},\mathsf{j}',\mathsf{k},\mathsf{k}') \\
\mathsf{dlseg}\ \alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') &\Leftrightarrow \exists \mathsf{j}'.\ \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k}',\mathsf{j}') * \mathsf{k}' \mapsto \mathsf{b},\mathsf{k},\mathsf{j}' \\
\mathsf{dlist}\ \alpha\,(\mathsf{i},\mathsf{j}') &\stackrel{\mathsf{def}}{=} \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathbf{nil},\mathsf{j}').
\end{aligned}$$

One can also define a doubly-linked list by

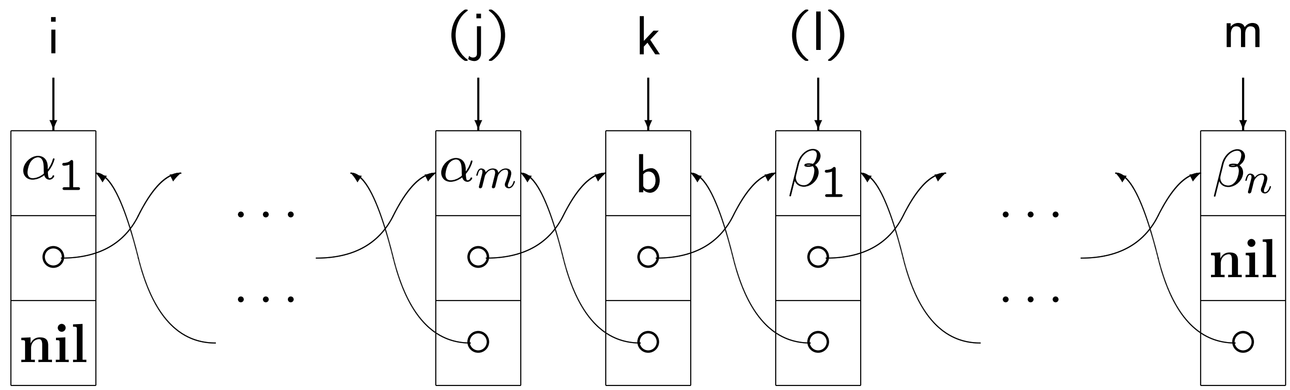
$$\mathsf{dlist}\ \alpha\,(\mathsf{i},\mathsf{j}') = \mathsf{dlseg}\ \alpha\,(\mathsf{i},\mathbf{nil},\mathbf{nil},\mathsf{j}').$$

Emptyness Conditions

$$\mathsf{dlseg}\,\alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil} \wedge \mathsf{i}' = \mathsf{j}'))$$
$$\mathsf{dlseg}\,\alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{j}' = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{i}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j}))$$
$$\mathsf{dlseg}\,\alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon)$$
$$\mathsf{dlseg}\,\alpha\,(\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \Rightarrow (\mathsf{i}' \neq \mathsf{j}' \Rightarrow \alpha \neq \epsilon).$$

(One can also define nontouching segments.)

Deleting an Element from a Doubly-Linked List



$\{\exists \mathsf{j}, \mathsf{l}.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathsf{l} := [\mathsf{k}+1]\ ;\ \mathsf{j} := [\mathsf{k}+2]\ ;$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{dispose}\ \mathsf{k}\ ;\ \mathbf{dispose}\ \mathsf{k}+1\ ;\ \mathbf{dispose}\ \mathsf{k}+2\ ;$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{if}\ \mathsf{j} = \mathbf{nil}\ \mathbf{then}$

$\quad \{\mathsf{i} = \mathsf{k} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon \wedge \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\quad \mathsf{i} := \mathsf{l}$

$\quad \{\mathsf{i} = \mathsf{l} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon \wedge \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathbf{else}$

$\quad \{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlseg}\ \alpha'\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{k}, \mathsf{n}$

$\qquad *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$

$\quad [\mathsf{j}+1] := \mathsf{l}\ ;$

$\quad \{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlseg}\ \alpha'\ (\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{l}, \mathsf{n}$

$\qquad *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\vdots$

$\vdots$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

if $\mathsf{l} = \mathbf{nil}$ **then**

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{k} = \mathsf{m} \wedge \beta = \epsilon\}$

$\mathsf{m} := \mathsf{j}$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{j} = \mathsf{m} \wedge \beta = \epsilon\}$

else

$\{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{k}$
$\quad *\ \mathsf{dlseg}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a}{\cdot}\beta'\}$

$[\mathsf{l} + 2] := \mathsf{j}$

$\{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{j}$
$\quad *\ \mathsf{dlseg}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a}{\cdot}\beta'\}$

$\{\mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{l}, \mathsf{j}, \mathbf{nil}, \mathsf{m})\}$

$\{\mathsf{dlseg}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})\}$

Exercise 5

When

$$\exists \alpha, \beta.\ (\mathsf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{lseg}\ \beta\ (\mathsf{j}, \mathsf{k})) \wedge \gamma = \alpha{\cdot}\beta,$$

we say that j is an *interior pointer* of the list segment described by $\mathsf{lseg}\ \gamma\ (\mathsf{i}, \mathsf{k})$.

1. Give an assertion describing a list segment with two interior pointers j_1 and j_2, such that j_1 comes before than, or at the same point as, j_2 in the ordering of the elements of the list segment.
2. Give an assertion describing a list segment with two interior pointers j_1 and j_2, where there is no constraint on the relative positions of j_1 and j_2.
3. Prove that the first assertion implies the second.

Exercise 6

A *braced list segment* is a list segment with an interior pointer j to its last element; in the special case where the list segment is empty, j is **nil**. Formally,

$$\mathsf{brlseg}\,\epsilon\,(\mathsf{i},\mathsf{j},\mathsf{k}) \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{k} \wedge \mathsf{j} = \mathbf{nil}$$

$$\mathsf{brlseg}\,\alpha{\cdot}a\,(\mathsf{i},\mathsf{j},\mathsf{k}) \stackrel{\text{def}}{=} \mathbf{lseg}\,\alpha\,(\mathsf{i},\mathsf{j}) \;*\; \mathsf{j} \mapsto \mathsf{a},\mathsf{k}.$$

Prove the assertion

$$\mathsf{brlseg}\,\alpha\,(\mathsf{i},\mathsf{j},\mathsf{k}) \Rightarrow \mathbf{lseg}\,\alpha\,(\mathsf{i},\mathsf{k}).$$

Exercise 7

Write nonrecursive procedures for manipulating braced list segments, that satisfy the following hypotheses. In each case, give an annotated specification of the body that proves it is a correct implementation of the procedure. In a few cases, you may wish to use the procedures defined in previous cases.

1. A procedure for looking up the final pointer:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{lookuppt}(\mathsf{k}; \mathsf{i}, \mathsf{j})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0) \wedge \mathsf{k} = \mathsf{k}_0\}.$$

   (This procedure should not alter the heap.)

2. A procedure for setting the final pointer:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{setpt}(\mathsf{i}; \mathsf{j}, \mathsf{k})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k})\}.$$

   (This procedure should not allocate or deallocate heap storage.)

3. A procedure for appending an element on the left:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{appleft}(\mathsf{i}, \mathsf{j}; \mathsf{a})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \mathsf{a}\cdot\alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}.$$

4. A procedure for deleting an element on the left:

   $$\{\mathsf{brlseg}\ \mathsf{a}\cdot\alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{delleft}(\mathsf{i}, \mathsf{j};\ )\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}.$$

5. A procedure for appending an element on the right:

   $$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}\ \mathsf{appright}(\mathsf{i}, \mathsf{j}; \mathsf{a})\{\alpha, \mathsf{k}_0\}\ \{\mathsf{brlseg}\ \alpha\cdot\mathsf{a}\ (\mathsf{i}, \mathsf{j}, \mathsf{k}_0)\}.$$

6. A procedure for concatenating two segments:

$$\{\mathsf{brlseg}\ \alpha\ (\mathsf{i}, \mathsf{j}, \mathsf{k_0}) * \mathsf{brlseg}\ \beta\ (\mathsf{i'}, \mathsf{j'}, \mathsf{k'_0})\}$$
$$\mathsf{conc}(\mathsf{i}, \mathsf{j}; \mathsf{i'}, \mathsf{j'})\{\alpha, \beta, \mathsf{k_0}, \mathsf{k'_0}\}$$
$$\{\mathsf{brlseg}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathsf{j}, \mathsf{k'_0})\}.$$

(This procedure should not allocate or deallocate heap storage.)

AN INTRODUCTION TO

SEPARATION LOGIC

5. Trees and Dags

John C. Reynolds
Carnegie Mellon University
Marktoberdorf August 15, 2008

S-expressions (a la LISP)

$\tau \in$ S-exps iff $\tau \in$ Atoms

or $\tau = (\tau_0 \cdot \tau_1)$ where $\tau_0, \tau_1 \in$ S-exps.

Representing S-expressions by Trees

For $\tau \in$ S-exps, we define the assertion $\mathsf{tree}\,\tau\,(i)$ by structural induction:

$\mathsf{tree}\,a\,(i)$ iff $\mathbf{emp} \wedge i = a$ when a is an atom

$\mathsf{tree}\,(\tau_0 \cdot \tau_1)\,(i)$ iff

$$\exists i_0, i_1.\ i \mapsto i_0, i_1 \ast \mathsf{tree}\,\tau_0\,(i_0) \ast \mathsf{tree}\,\tau_1\,(i_1).$$

One can show that the assertions $\mathsf{tree}\,\tau\,(i)$ and $\exists\tau.\ \mathsf{tree}\,\tau\,(i)$ are precise.

Copying Trees

We will show that

$$\begin{aligned}
&\mathsf{copytree}(\mathsf{j};\mathsf{i}) = \\
&\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else} \\
&\qquad \mathbf{newvar}\ \mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1\ \mathbf{in} \\
&\qquad\quad \big(\mathsf{i}_0 := [\mathsf{i}]\ ;\ \mathsf{i}_1 := [\mathsf{i}+1]\ ; \\
&\qquad\quad \mathsf{copytree}(\mathsf{j}_0;\mathsf{i}_0)\ ;\ \mathsf{copytree}(\mathsf{j}_1;\mathsf{i}_1)\ ;\ \mathsf{j} := \mathbf{cons}(\mathsf{j}_0,\mathsf{j}_1)\big).
\end{aligned}$$

satisfies

$$\{\mathsf{tree}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau\}\ \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}.$$

$$\begin{aligned}
&\{\mathsf{tree}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau\}\ \{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\} \vdash \\
&\qquad \{\mathsf{tree}\,\tau(\mathsf{i})\} \\
&\qquad\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then} \\
&\qquad\quad \{\mathsf{isatom}(\tau) \wedge \mathbf{emp} \wedge \mathsf{i} = \tau\} \\
&\qquad\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{i} = \tau))\} \\
&\qquad\quad \mathsf{j} := \mathsf{i} \\
&\qquad\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{j} = \tau))\} \\
&\qquad\qquad \vdots
\end{aligned}$$

$\vdots$

else

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{tree}\,(\tau_0 \cdot \tau_1)(\mathsf{i})\}$

newvar $\mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1$ **in**

$\big(\mathsf{i}_0 := [\mathsf{i}]\ ;\ \mathsf{i}_1 := [\mathsf{i}+1]\ ;$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1)$
$\quad\wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\ \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1))\}$

$$\left.\begin{array}{l}\{\mathsf{tree}\,\tau_0(\mathsf{i}_0)\}\\ \mathsf{copytree}(\mathsf{j}_0;\mathsf{i}_0)\{\tau_0\}\\ \{\mathsf{tree}\,\tau_0(\mathsf{i}_0)\ *\ \mathsf{tree}\,\tau_0(\mathsf{j}_0)\}\end{array}\right\}\ *\ \left[\begin{array}{l}\tau = (\tau_0 \cdot \tau_1) \wedge\\ (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\\ \quad \mathsf{tree}\,\tau_1\,(\mathsf{i}_1))\end{array}\right\}\exists \tau_0, \tau_1$$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *$
$\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_0\,(\mathsf{j}_0))\}$

$$\left.\begin{array}{l}\{\mathsf{tree}\,\tau_1(\mathsf{i}_1)\}\\ \mathsf{copytree}(\mathsf{j}_1;\mathsf{i}_1)\{\tau_1\}\\ \{\mathsf{tree}\,\tau(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_1(\mathsf{j}_1)\}\end{array}\right\}\ *\ \left[\begin{array}{l}\tau = (\tau_0 \cdot \tau_1) \wedge\\ (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\\ \quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\\ \quad \mathsf{tree}\,\tau_0\,(\mathsf{j}_0))\end{array}\right\}\exists \tau_0, \tau_1$$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *$
$\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_0\,(\mathsf{j}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\}$

$\mathsf{j} := \mathbf{cons}(\mathsf{j}_0, \mathsf{j}_1)$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\ \mathsf{j} \mapsto \mathsf{j}_0, \mathsf{j}_1\ *$
$\quad \mathsf{tree}\,\tau_0\,(\mathsf{i}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{i}_1)\ *\ \mathsf{tree}\,\tau_0\,(\mathsf{j}_0)\ *\ \mathsf{tree}\,\tau_1\,(\mathsf{j}_1))\}$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge (\mathsf{tree}\,(\tau_0{\cdot}\tau_1)\,(\mathsf{i}) * \mathsf{tree}\,(\tau_0{\cdot}\tau_1)\,(\mathsf{j}))\}\big)$

$\{\mathsf{tree}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}.$

Representing S-expressions by Dags

For $\tau \in$ S-exps, we define

$$\mathsf{dag}\,\tau\,(i)$$

by:

$$\mathsf{dag}\,a\,(i) \text{ iff } i = a \qquad \text{when } a \text{ is an atom}$$

$$\mathsf{dag}\,(\tau_0 \cdot \tau_1)\,(i) \text{ iff}$$
$$\exists i_0, i_1.\ i \mapsto i_0, i_1 \ * \ (\mathsf{dag}\,\tau_0\,(i_0) \wedge \mathsf{dag}\,\tau_1\,(i_1)).$$

Proposition 6 (1) $\mathsf{dag}\,\tau\,(i)$ *and* (2) $\exists\tau.\ \mathsf{dag}\,\tau\,(i)$ *are intuitionistic assertions.*

A Problem

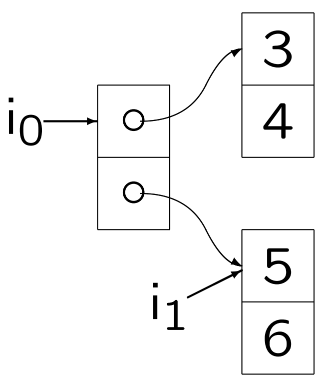
Suppose we wish to prove that

$$\{\mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{dag}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}$$

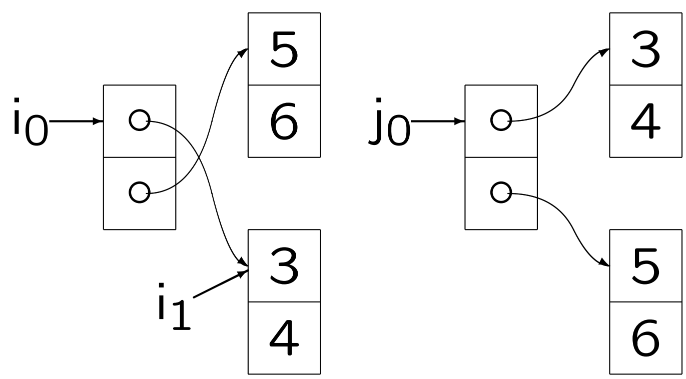
Then, we must use this specification as a hypothesis in proving that the first recursive call in the procedure body satisfies:

$$\begin{aligned}
&\{\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\ (\mathsf{dag}\ \tau_0(\mathsf{i}_0) \wedge \mathsf{dag}\ \tau_1(\mathsf{i}_1))\}\\
&\mathsf{copytree}(\mathsf{i}_0;\mathsf{j}_0)\\
&\{\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\ (\mathsf{dag}\ \tau_0(\mathsf{i}_0) \wedge \mathsf{dag}\ \tau_1(\mathsf{i}_1))\ *\ \mathsf{tree}\ \tau_0(\mathsf{j}_0)\}.
\end{aligned}$$

But the hypothesis is not strong enough to imply this. For example, suppose $\tau_0 = ((3{\cdot}4){\cdot}(5{\cdot}6))$ and $\tau_1 = (5{\cdot}6)$. Then $\mathsf{copytree}(\mathsf{i}_0;\mathsf{j}_0)$ might change the state from



into



where $\mathsf{dag}\ \tau_1\ (\mathsf{i}_1)$ is false.

Possible Solutions

1. Introduce ghost variables denoting heaps, e.g.

$$\{\mathbf{this}(\mathsf{h}_0) \wedge \mathsf{dag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathbf{this}(\mathsf{h}_0)\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}$$

2. Introduce ghost variables denoting assertions, e.g.

$$\{\mathsf{p} \wedge \mathsf{dag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{p}\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}$$

3. Introduce fractional permissions. Then one could define an assertion $\mathsf{passdag}\,\tau(\mathsf{i})$ describing a read-only heap containing a dag, and use it to specify:

$$\{\mathsf{passdag}\,\tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{passdag}\,\tau(\mathsf{i})\ *\ \mathsf{tree}\,\tau(\mathsf{j})\}.$$

We will explore the second approach.

Assertion Variables

We extend the concept of state to include an *assertion store* mapping assertion variables into properties of heaps:

$$\begin{aligned} \text{AStores}_A &= A \to (\text{Heaps} \to \mathbf{B}) \\ \text{States}_{AV} &= \text{AStores}_A \times \text{Stores}_V \times \text{Heaps}, \end{aligned}$$

where A denotes a finite set of *assertion variables*.

Assertion stores have no effect on the execution of commands, but they affect the meaning of assertions. Thus we write

$$as, s, h \vDash p$$

(instead of $s, h \vDash p$) to indicate that the state as, s, h *satisfies* p.

Then, when an assertion variable is used as an assertion:

$$as, s, h \vDash a \text{ iff } as(a)(h).$$

The Substitution Rule Revisited

- Substitution (SUB)

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/a_1 \to p_1, \ldots, a_m \to p_m, v_1 \to e_1, \ldots, v_n \to e_n}$$

  where $a_1, \ldots, a_m$ are the assertion variables occurring free in p or q, $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j or in any p_j.

In $\{\mathsf{a}\}\ \mathsf{x} := \mathsf{y}\ \{\mathsf{a}\}$, we can substitute $\mathsf{a} \to \mathsf{y} = \mathsf{z}, \mathsf{x} \to \mathsf{x}, \mathsf{y} \to \mathsf{y}$ to obtain

$$\{\mathsf{y} = \mathsf{z}\}\ \mathsf{x} := \mathsf{y}\ \{\mathsf{y} = \mathsf{z}\},$$

but we cannot substitute $\mathsf{a} \to \mathsf{x} = \mathsf{z}, \mathsf{x} \to \mathsf{x}, \mathsf{y} \to \mathsf{y}$ to obtain

$$\{\mathsf{x} = \mathsf{z}\}\ \mathsf{x} := \mathsf{y}\ \{\mathsf{x} = \mathsf{z}\}.$$

Copying Dags to Trees

We will prove that the procedure

$$\begin{array}{l}\mathsf{copytree}(\mathsf{i};\mathsf{j}) = \\ \quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else} \\ \qquad \mathbf{newvar}\ \mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1\ \mathbf{in} \\ \qquad\quad \big(\mathsf{i}_0 := [\mathsf{i}]\ ;\ \mathsf{i}_1 := [\mathsf{i}+1]\ ; \\ \qquad\quad \mathbf{copytree}(\mathsf{i}_0;\mathsf{j}_0);\mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1);\mathsf{j}{:=}\mathbf{cons}(\mathsf{j}_0,\mathsf{j}_1)\big)\end{array}$$

satisfies

$$\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{p}\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}.$$

We can take p to be $\mathsf{dag}\ \tau(\mathsf{i})$, to obtain the specification

$$\{\mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{dag}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\},$$

but this is too weak to serve as a recursion hypothesis.

$$\begin{array}{l}\{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{j};\mathsf{i})\{\tau,\mathsf{p}\}\ \{\mathsf{p}\ *\ \mathsf{tree}\ \tau(\mathsf{j})\} \vdash \\ \qquad \{\mathsf{p} \wedge \mathsf{dag}\ \tau(\mathsf{i})\} \\ \qquad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then} \\ \qquad\quad \{\mathsf{p} \wedge \mathsf{isatom}(\tau) \wedge \tau = \mathsf{i}\} \\ \qquad\quad \{\mathsf{p}\ *\ (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{i} \wedge \mathbf{emp})\} \\ \qquad\quad \mathsf{j} := \mathsf{i} \\ \qquad\quad \{\mathsf{p}\ *\ (\mathsf{isatom}(\tau) \wedge \tau = \mathsf{j} \wedge \mathbf{emp})\} \\ \qquad\qquad \vdots\end{array}$$

$\vdots$

else

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\ (\tau_0 \cdot \tau_1)(\mathsf{i})\}$

newvar $\mathsf{i}_0, \mathsf{i}_1, \mathsf{j}_0, \mathsf{j}_1$ **in**

$\big(\mathsf{i}_0 := [\mathsf{i}]\ ;\ \mathsf{i}_1 := [\mathsf{i} + 1]\ ;$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge$
$\quad \mathsf{p} \wedge (\mathsf{i} \mapsto \mathsf{i}_0, \mathsf{i}_1\ *\ (\mathsf{dag}\, \tau_0\, (\mathsf{i}_0) \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1)))\}$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge$
$\quad \mathsf{p} \wedge (\mathbf{true}\ *\ (\mathsf{dag}\, \tau_0\, (\mathsf{i}_0) \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1)))\}$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge$
$\quad \mathsf{p} \wedge ((\mathbf{true}\ *\ \mathsf{dag}\, \tau_0\, (\mathsf{i}_0)) \wedge (\mathbf{true}\ *\ \mathsf{dag}\, \tau_1\, (\mathsf{i}_1)))\}$

$\{\exists \tau_0, \tau_1.\ \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1) \wedge \mathsf{dag}\, \tau_0\, (\mathsf{i}_0)\}$

$$\left.\begin{array}{l} \{\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1) \wedge \mathsf{dag}\, \tau_0(\mathsf{i}_0)\} \\ \mathsf{copytree}(\mathsf{j}_0; \mathsf{i}_0)\{\tau_0, \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1)\} \\ \{(\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1))\ *\ \mathsf{tree}\, \tau_0(\mathsf{j}_0)\} \end{array}\right\} \exists \tau_0, \tau_1$$

$\{\exists \tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_1\, (\mathsf{i}_1))\ *\ \mathsf{tree}\, \tau_0\, (\mathsf{j}_0)\}$

$$\left.\left.\begin{array}{l} \{\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p} \wedge \mathsf{dag}\, \tau_1(\mathsf{i}_1)\} \\ \mathsf{copytree}(\mathsf{j}_1; \mathsf{i}_1)\{\tau_1, \tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p}\} \\ \{(\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p})\ *\ \mathsf{tree}\, \tau_1(\mathsf{j}_1)\} \end{array}\right\} *\ \mathsf{tree}\, \tau_0\, (\mathsf{j}_0)\right\} \exists \tau_0, \tau_1$$

$\{\exists \tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p})\ *\ \mathsf{tree}\, \tau_0\, (\mathsf{j}_0)\ *\ \mathsf{tree}\, \tau_1\, (\mathsf{j}_1)\}$

$\mathsf{j} := \mathbf{cons}(\mathsf{j}_0, \mathsf{j}_1)$

$\{\exists \tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p})\ *$
$\quad \mathsf{j} \mapsto \mathsf{j}_0, \mathsf{j}_1\ *\ \mathsf{tree}\, \tau_0\, (\mathsf{j}_0)\ *\ \mathsf{tree}\, \tau_1\, (\mathsf{j}_1)\}$

$\{\exists \tau_0, \tau_1.\ (\tau = (\tau_0 \cdot \tau_1) \wedge \mathsf{p})\ *\ \mathsf{tree}\, (\tau_0 \cdot \tau_1)\, (\mathsf{j})\}\big)$

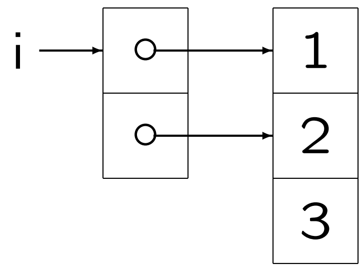
$\{\mathsf{p}\ *\ \mathsf{tree}\, \tau(\mathsf{j})\}$

Skewed Sharing

Our definition of dag permits *skewed sharing*. For example,

$$\mathsf{dag}\,((1 \cdot 2) \cdot (2 \cdot 3))\,(\mathsf{i})$$

holds when



Skewed sharing is not a problem for the algorithms we have seen so far, which only examine dags while ignoring their sharing structure. But it causes difficulties with algorithms that modify dags or depend upon the sharing structure.

A Possible Solution

- We add to the state a mapping ϕ from the domain of the heap to natural numbers, called the *field count*.
- When $x := \mathbf{cons}(e_1, \ldots, e_n)$ sets x to the address a. the field count is extended so that

$$\phi(a) = n \qquad \phi(a+1) = 0 \quad \cdots \quad \phi(a+n-1) = 0.$$

- We introduce the assertion $e \stackrel{[\hat{e}]}{\mapsto} e'$, with the meaning

$$\begin{aligned} s, h, \phi \vDash e \stackrel{[\hat{e}]}{\mapsto} e' \text{ iff} \\ \operatorname{dom} h = \{[\![e]\!]_{\mathsf{exp}} s\} \text{ and } h([\![e]\!]_{\mathsf{exp}} s) = [\![e']\!]_{\mathsf{exp}} s \\ \text{and } \phi([\![e]\!]_{\mathsf{exp}} s) = [\![\hat{e}]\!]_{\mathsf{exp}} s. \end{aligned}$$

- We also introduce the following abbreviations:

$$e \stackrel{[\hat{e}]}{\mapsto} - \stackrel{\text{def}}{=} \exists x'.\, e \stackrel{[\hat{e}]}{\mapsto} x' \quad \text{where } x' \text{ not free in } e$$

$$e \stackrel{[\hat{e}]}{\hookrightarrow} e' \stackrel{\text{def}}{=} e \stackrel{[\hat{e}]}{\mapsto} e' * \mathbf{true}$$

$$e \stackrel{!}{\mapsto} e_1, \ldots, e_n \stackrel{\text{def}}{=} e \stackrel{[n]}{\mapsto} e_1 * e{+}1 \stackrel{[0]}{\mapsto} e_2 * \cdots * e{+}n{-}1 \stackrel{[0]}{\mapsto} e_n$$

$$\begin{aligned} e \stackrel{!}{\hookrightarrow} e_1, \ldots, e_n &\stackrel{\text{def}}{=} e \stackrel{[n]}{\hookrightarrow} e_1 * e{+}1 \stackrel{[0]}{\hookrightarrow} e_2 * \cdots * e{+}n{-}1 \stackrel{[0]}{\hookrightarrow} e_n \\ &\text{iff } e \stackrel{!}{\mapsto} e_1, \ldots, e_n * \mathbf{true}. \end{aligned}$$

Axiom Schema

$$e \overset{[\mathsf{n}]}{\mapsto} e' \Rightarrow e \mapsto e'$$

$$e \overset{[\mathsf{m}]}{\hookrightarrow} - \wedge e \overset{[\mathsf{n}]}{\hookrightarrow} - \Rightarrow \mathsf{m} = \mathsf{n}$$

$$2 \leq \mathsf{k} \leq \mathsf{n} \wedge e \overset{[\mathsf{n}]}{\hookrightarrow} - \wedge e + \mathsf{k} - 1 \hookrightarrow - \Rightarrow e + \mathsf{k} - 1 \overset{[0]}{\hookrightarrow} -$$

$$e \overset{!}{\hookrightarrow} e_1, \ldots, e_m \wedge e' \overset{!}{\hookrightarrow} e'_1, \ldots, e'_n \wedge e \neq e' \Rightarrow$$
$$e \overset{!}{\mapsto} e_1, \ldots, e_m \ast e' \overset{!}{\mapsto} e'_1, \ldots, e'_n \ast \mathbf{true}.$$

(The last of these axiom schemas makes it clear that skewed sharing has been prohibited.)

Additional Inference Rules

- Allocation: local nonoverwriting form (FCCONSNOL)

$$\overline{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \overset{!}{\mapsto} \overline{e}\},}$$

  where $v \notin \mathrm{FV}(\overline{e})$.

- Mutation: local form (FCMUL)

$$\overline{\{e \overset{[\widehat{e}]}{\mapsto} -\}\ [e] := e'\ \{e \overset{[\widehat{e}]}{\mapsto} e'\}.}$$

- Lookup: local nonoverwriting form (FCLKNOL)

$$\overline{\{e \overset{[\widehat{e}]}{\mapsto} v''\}\ v := [e]\ \{d\}v = v'' \wedge (e \overset{[\widehat{e}]}{\mapsto} v),}$$

  where $v \notin \mathrm{FV}(e, \widehat{e})$.

A Problem with Deallocation

If one can deallocate single fields, the use of field counts can be disrupted by deallocating a part of record. For example,

$$\mathsf{j} := \mathbf{cons}(1,2)\,;\mathbf{dispose}\;\mathsf{j}+1\,;\mathsf{k} := \mathbf{cons}(3,4)\,;\mathsf{i} := \mathbf{cons}(\mathsf{j},\mathsf{k})$$

could produce skewed sharing if the new record allocated by the second **cons** were placed at locations $\mathsf{j}+1$ and $\mathsf{j}+2$.

A solution is to replace **dispose** e with an command **dispose** (e, n) that disposes of an entire n-field record — and then to require that this record must have been created by an execution of **cons**:

- The local form (FCDISL)

$$\frac{}{\{e \overset{!}{\mapsto} -^n\}\ \mathbf{dispose}\,(e,n)\ \{\mathbf{emp}\}.}$$

- The global (and backward-reasoning) form (FCDISG)

$$\frac{}{\{(e \overset{!}{\mapsto} -^n) \ *\ r\}\ \mathbf{dispose}\,(e,n)\ \{r\}.}$$

(Here $-^n$ denotes a list of n occurrences of $-$.)

Exercise 9

If τ is an S-expression, then $|\tau|$, called the *flattening* of τ, is the sequence defined by:

$$|a| = [a] \quad \text{when } a \text{ is an atom}$$
$$|(t_0 \cdot t_1)| = |\tau_0| \cdot |\tau_1|.$$

Here $[a]$ denotes the sequence whose only element is a, and the "$\cdot$" on the right of the last equation denotes the concatenation of sequences.

Define and prove correct (by an annotated specification of its body) a recursive procedure $\mathsf{flatten}$ that mutates a tree denoting an S-expression τ into a singly-linked list segment denoting the flattening of τ. This procedure should not do any allocation or disposal of heap storage. However, since a list segment representing $|\tau|$ contains one more two-cell than a tree representing τ, the procedure should be given as input, in addition to the tree representing τ, a single two-cell, which will become the initial cell of the list segment that is constructed.

More precisely, the procedure should satisfy

$$\{\mathsf{tree}\,\tau\,(\mathsf{i}) \ast \mathsf{j} \mapsto -,-\}$$
$$\mathsf{flatten}(\mathsf{i},\mathsf{j},\mathsf{k})$$
$$\{\mathsf{lseg}\,|\tau|\,(\mathsf{j},\mathsf{k})\}.$$

(Note that $\mathsf{flatten}$ must not assign to the variables i, j,